Магда Ю. С.

КОМПЬЮТЕР В ДОМАШНЕЙ ЛАБОРАТОРИИ



Москва, 2008

УДК 621.396.6
ББК 32.872
М12

М12 Магда Ю. С.
Компьютер в домашней лаборатории. – М.: ДМК Пресс, 2008. – 200 с.: ил.

ISBN 978-5-94074-420-7

В книге рассматривается широкий круг вопросов, связанных с практическим применением персональных компьютеров, работающих под управлением операционных систем Windows, для создания устройств домашней электроники. Материал книги охватывает многочисленные аппаратно-программные аспекты проектирования любительских электронных устройств, управляемых от параллельного, последовательного интерфейсов персонального компьютера и звуковой карты. Значительное внимание уделено новым технологиям USB и Bluetooth, а также возможностям их применения в любительской практике. В книге в доступной форме излагается материал по разработке несложных драйверов устройств пользователя, что существенно расширяет возможности их проектирования пользователями различного уровня подготовки. Приводятся многочисленные примеры разработки несложных аппаратно-программных систем сбора аналоговой и цифровой информации, измерительных систем, систем управления внешними устройствами и т. д.

Издание может быть полезно радиолюбителям различного уровня подготовки и всем, кто интересуется компьютерными системами управления и контроля.

УДК 621.396.6
ББК 32.872

ISBN 978-5-94074-420-7

Оглавление

Введение

Персональные компьютеры применяются настолько широко, что, казалось бы, найти им новое применение в настоящее время не так и просто. Тем не менее, есть несколько сфер человеческой деятельности, где персональный компьютер только в последнее время стал завоевывать серьезные позиции. Одна из таких сфер – домашняя компьютерная электроника или, по-другому, использование ПК для создания собственных аппаратно-программных систем, способных выполнять самые разнообразные функции под управлением компьютера. Эта область включает не только создание различных робототехнических систем, но и устройств измерения, сигнализации и управления.

Эта книга посвящена практическим аспектам разработки систем компьютерной электроники, работающих под управлением операционных систем Windows на основе программно-аппаратных устройств, разработанных автором.

Литературы и документации по данной тематике мало, поскольку раскрытие этой темы сопряжено со значительными трудностями, связанными с тем, что охватывается очень широкий диапазон знаний – от элементов аналоговой и цифровой схемотехники до программирования USB и Bluetooth. Предлагаемая вашему вниманию книга призвана восполнить этот пробел.

Читатели без особого труда смогут адаптировать и усовершенствовать приведенный в книге программный код и схемотехнические решения при разработке собственных систем компьютерной электроники.

Книга рассчитана на широкий круг читателей – от начинающих до опытных пользователей.

Структура книги

Структура книги такова, что материал можно изучать выборочно, отдельными главами или последовательно, начиная с первой главы. Это позволяет различным категориям читателей изучать тот материал, который им более всего интересен.

Книга состоит из 6 глав; краткий обзор каждой из них:

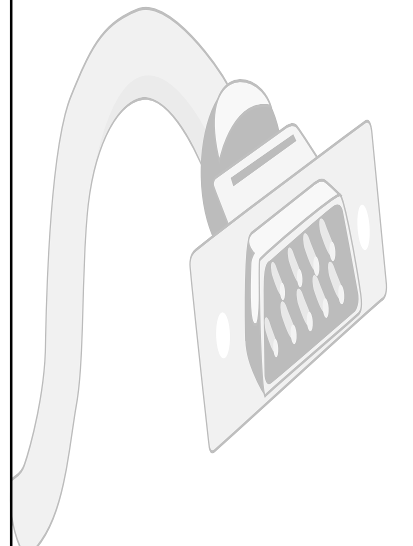
- глава 1 «Возможности персонального компьютера». В этой главе дается обзор основных вариантов применения персонального компьютера в системах домашней электроники;
- глава 2 «Архитектура ввода-вывода». Материал этой главы посвящен вопросам архитектуры подсистемы ввода-вывода персональных компьютеров. Рассматриваются



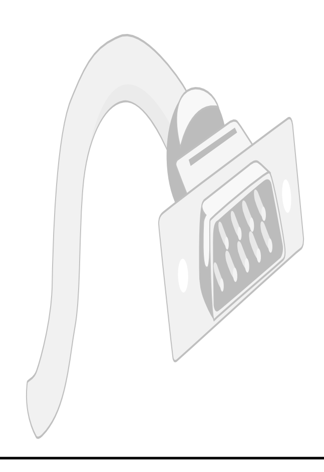
общие вопросы функционирования шинных интерфейсов, схемотехники и программирования устройств ввода-вывода пользователя;

- глава 3 «Параллельный порт в лабораторных разработках». В этой главе детально проанализированы принципы функционирования параллельного порта персонального компьютера и его программирование. Рассматриваются многочисленные аппаратно-программные проекты систем измерения и управления с управлением от параллельного порта ПК в операционных системах Windows;
- глава 4 «Последовательный порт персонального компьютера в любительских разработках». Эта глава содержит материал по аппаратной архитектуре, протоколам обмена и программированию последовательного порта персонального компьютера. Приводятся практические проекты аппаратно-программных систем с использованием последовательного порта;
- глава 5 «Звуковые карты и их применение». Материал главы посвящен вопросам разработки и программирования систем домашней электроники на основе звуковой карты. В главе проанализированы основы программирования генераторов частот и систем управления с использованием библиотеки DirectSound пакета DirectX;
- глава 6 «Интерфейсы USB и Bluetooth». В этой главе рассматривается широкий круг вопросов, связанных с применением устройств USB и Bluetooth, включая основы их функционирования и элементы программирования.

Автор благодарит коллектив издательства «ДМК» за помощь при подготовке книги к изданию. Особая признательность жене Юлии за поддержку и помощь при написании книги.



Возможности персонального компьютера



1 Возможности персонального компьютера

Персональные компьютеры в настоящее время широко используются практически во всех сферах деятельности человека, которые известны всем и каждому: образование, экономика, научные исследования, коммуникации, индустрия развлечений. Тем не менее, есть и более специализированные сферы деятельности, в которых компьютер может найти и находит достаточно эффективное применение, например, системы сбора и обработки информации в науке и промышленности, робототехника и системы управления, домашняя электроника. Как раз использование персонального компьютера в различных электронных устройствах, а также создание небольших систем сбора и обработки информации, простых систем охранной сигнализации и многих других устройств обсуждается в этой книге.

Создание собственных аппаратно-программных проектов, в основе которых лежит использование домашнего персонального компьютера, требует определенных знаний, как архитектуры самого ПК, так и определенных знаний и навыков программирования. Кроме того, разработка таких систем требует определенного уровня знаний в схемотехнике, как в аналоговой, так и в цифровой. Тем не менее, рассмотренные в книге проекты под силу реализовать даже пользователям средней руки. Проектирование собственных устройств с использованием ПК базируется на применении возможностей аппаратных средств, входящих в состав компьютера:

- параллельного порта принтера;
- последовательного порта;
- звуковой карты;
- устройств USB и Bluetooth.

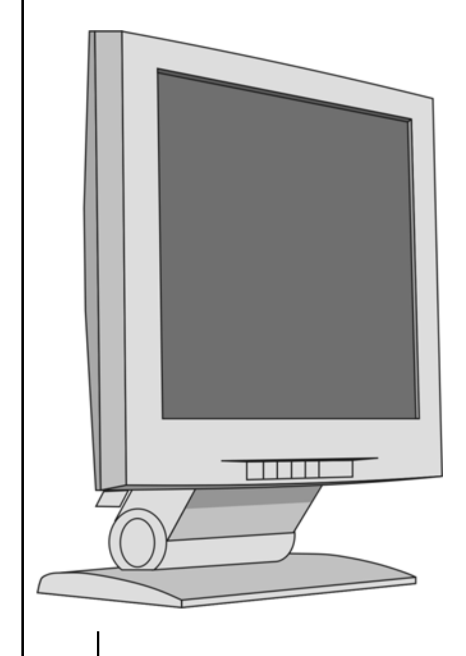
Все эти устройства, помимо их стандартного применения, позволяют создавать и проекты домашней электроники, которые могут, в принципе, не уступать даже лабораторным и промышленным системам. Более того, в мире выпускается очень много оборудования, применение которого базируется на использовании вышеперечисленных аппаратных средств. Достаточно вспомнить многочисленные версии «электронных осциллографов», базирующихся на применении звуковой карты и протокола USB, системы сбора данных на базе последовательного и параллельного портов, системы удаленного управления на базе технологии Bluetooth.

В последнее время ведущие разработчики программного обеспечения, в их числе и Microsoft, начали создание программного обеспечения, направленного на поддержку различных аппаратно-программных систем, которые может разработать любой пользователь, используя то или иное периферийное оборудование. Например, известный и быстро развивающийся программный продукт, такой, как Microsoft Robotics Studio, предлагает программный интер-

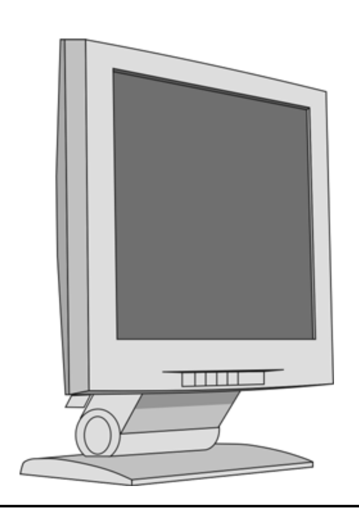
фейс, позволяющий создавать системы управления роботами и другими устройствами домашней электроники. Фирма Phidgets и целый ряд других фирм выпускают электронные модули на базе USB, которые работают с Robotics Studio, снабжены программным интерфейсом и легко интегрируются в ПК. С помощью таких устройств можно создавать системы управления и измерения.

Этот перечень можно продолжить. Интерес пользователей к разработкам собственных аппаратно-программных проектов с каждым днем возрастает, и с каждым днем на рынке появляется все больше и больше устройств, подобных тем, которые выпускает Phidgets. Более того, появление и быстрое развитие беспроводных технологий открывает новые горизонты для творчества. Кстати, в этой нише в последнее время появилось также много фирм, выпускающих аппаратно-программные модули для создания пользовательских электронных систем. К сожалению, такие системы являются относительно дорогими для отечественных пользователей, чтобы их можно было легко использовать в собственных электронных проектах. Предлагаемые в книге проекты не требуют больших финансовых затрат на комплектующие, а используемое в них программное обеспечение является бесплатным. Такие аппаратно-программные проекты позволяют решать довольно серьезные задачи.

Большинство читателей хорошо представляют себе архитектуру персонального компьютера, но, возможно, не знакомы с тем, как взаимодействуют устройства ввода-вывода (а к ним относятся практически все устройства, кроме памяти) с процессором. В следующей главе мы рассмотрим принципы функционирования устройств ввода-вывода, поскольку именно они будут применяться для разработки наших проектов.



Архитектура ввода-вывода



2 Архитектура ввода-вывода

В этой главе вкратце рассматриваются основные аппаратно-программные аспекты, касающиеся организации и функционирования подсистем ввода-вывода в компьютерных системах с различной конфигурацией системных шин, включая шину PCI платформы x86, которая наиболее часто применяется в персональных компьютерах. Такой анализ поможет глубже понять принципы функционирования различных периферийных устройств в различных системах и облегчит задачи программирования таких устройств. Под термином «подсистема ввода-вывода» обычно понимают как аппаратные устройства компьютера, так и программный интерфейс, позволяющий взаимодействовать с ними.

Операции ввода-вывода для большинства компьютерных систем отличаются от операций с памятью, как на уровне инструкций процессора, так и в плане схемотехнической реализации самих устройств. Схему ввода-вывода информации можно представить себе так, как показано на рис. 2.1.

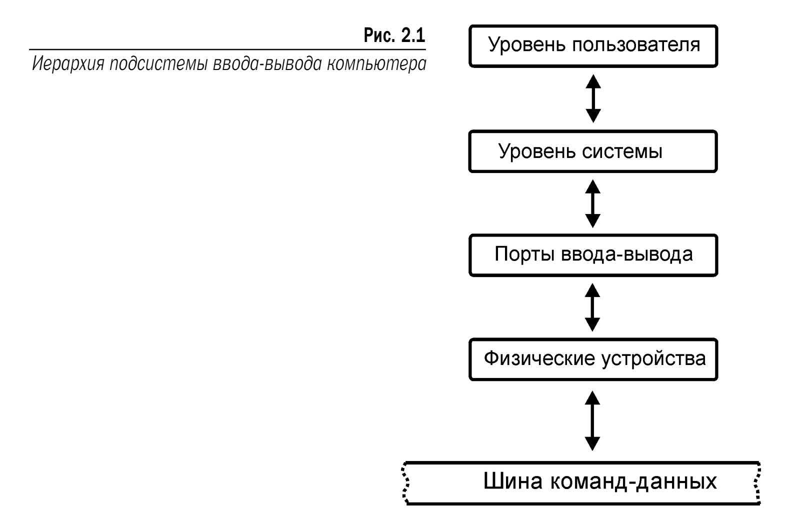


Рис. 2.1
Иерархия подсистемы ввода-вывода компьютера

На самом нижнем уровне устройства взаимодействуют с аппаратурой компьютера через одну или несколько шин команд-данных, интерфейс которых для разных аппаратных платформ вообще отличается, хотя некоторые стандарты (например, PCI PCI Express и т. д.) используются разными платформами.

Взаимодействие устройств на уровне шин осуществляется с помощью электрических сигналов, временные характеристики которых соответствуют используемому стандарту, а аппаратный интерфейс физического устройства должен обеспечить генерацию всех необходимых сигналов обмена.

Обмен данными с устройством осуществляется посредством портов ввода-вывода и отображаемой памяти. В этом случае все операции с устройством выполняются на уровне инструкций процессора или микроконтроллера. Термин «вывод данных» означает операцию записи данных в устройство, а термин «ввод данных» означает чтение данных из устройства. Далее мы будем использовать эти термины как синонимы.

Схемотехническая реализация портов ввода-вывода зависит от временной диаграммы работы сигнальных линий процессора и реализации алгоритма обмена. Напомню, что центральным устройством любой компьютерной системы является процессор или микроконтроллер, который формирует сигналы обмена данными с устройствами в соответствии с определенными правилами.

Для стандартизации обмена данными процессора и устройств ввода-вывода используется целый ряд промышленных стандартов обмена данными (так называемых «шинных интерфейсов» или просто шин), наиболее известными из которых являются PCI и PCI Express. Шинный интерфейс предоставляет разработчикам устройств и пользователям единый стандарт обмена данными, что значительно облегчает интеграцию различных устройств в систему. Физически шина представляет собой набор электрических линий, сигналы на которых подчиняются определенным правилам или стандартам, например, стандарту PCI, откуда и название шины.

С другой стороны, в большинстве случаев набор сигналов процессора или контроллера не соответствует стандартам шины, поэтому между процессором и шиной помещают дополнительное устройство, которое обычно называют контроллером шины. Контроллер шины выполняет функции преобразования сигналов процессора в сигналы шинного интерфейса. Например, для систем с процессорами Intel контроллер шины преобразует сигналы процессора в стандарт PCI или PCI Express (рис. 2.2).

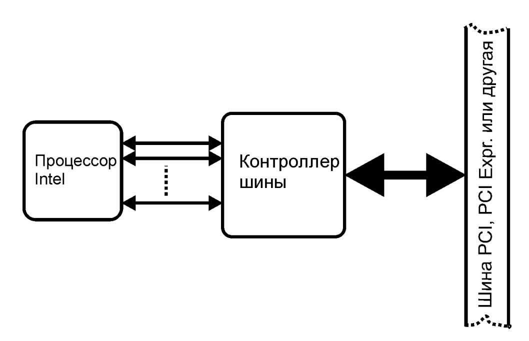


Рис. 2.2
Упрощенная структура шинной архитектуры

На этом рисунке контроллер шины для упрощения показан в виде одного модуля, хотя на самом деле он может включать несколько интегральных микросхем и быть довольно сложным. Шина PCI в данном примере является системной шиной, и к ней могут подключаться другие устройства, сигналы которых соответствуют стандарту данной шины. Например, к шине PCI может подключаться хост-контроллер шины USB, который, в свою очередь, будет формировать сигналы синхронизации на шине USB для подключения USB-устройств.

Таким образом, в любой более-менее сложной компьютерной системе обмен данными между процессором и устройствами выполняется посредством одной или нескольких шин





команд/данных. В дальнейшем для краткости мы будем использовать термин «системная шина». Шина PCI далеко не единственный тип системной шины, используемый в компьютерных системах. Существует целый ряд других, широко распространенных промышленных стандартов (CANBUS, I2C, SPI и т. д.), которые весьма широко используются в различных компьютерных системах.

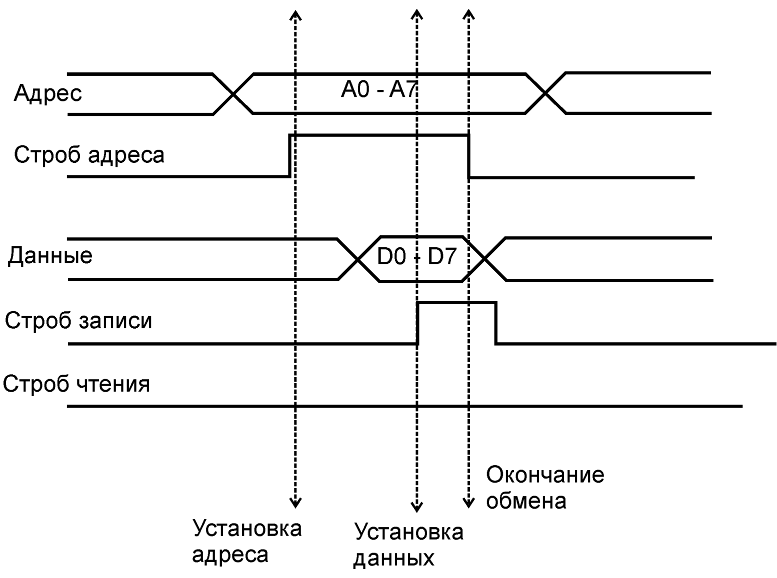
Как осуществляется обмен данными по шине? В компьютерной системе для выполнения обмена данными с любым устройством необходимо выполнить стандартную последовательность шагов:

- установить на адресных линиях системной шины или шины команд/данных адрес устройства, к которому происходит обращение. При этом устройство, распознавшее адрес как собственный, должно выполнить необходимые переключения в схеме для подготовки обмена данными (дешифрация адреса, выбор/разрешение чтения/записи и т. д.);
- установить сигнал синхронизации адреса. Обычно это перепад напряжения на одной из управляющих линий системной шины, который предназначен для использования логикой устройства ввода-вывода;
- установить на линиях данных системной шины данные и сигнал синхронизации данных, сообщающий устройству, что данные действительны;
- закончить обмен данными, сняв адрес устройства с линий адреса и данные с линий данных.

Это весьма упрощенная схема функционирования большинства шин, но она дает представление о принципах их работы. Рассмотрим, как могли бы выполняться операции ввода-вывода данных при подключении устройства к гипотетической шине. Начнем с записи байта в устройство. Положим, что запись (вывод) байта осуществляется на шине с 8-ю линиями адресов и 8-ю линиями данных (рис. 2.3).

Рис. 2.3

Временная диаграмма записи данных в устройство на гипотетической шине





Наша гипотетическая шина имеет 8 линий адреса, 8 линий данных и две линии синхронизации записи и чтения данных.

Операцию обмена данными с устройством посредством сигналов шины называют «циклом шины». Обмен данными на нашей шине (см. рис. 2.3) начинается с установки адреса на линиях адреса A0–A7, для декодирования которого строб адреса устанавливается в высокий уровень. Цифровые схемы устройства на этом этапе должны обеспечить декодирование адреса. Если устройство распознает адрес на линиях адреса как «свой», то цифровая логика должна подготовить обмен данными (в данном случае, запись байта в устройство). При записи данных строб чтения на шине должен иметь низкий уровень.

Далее, по прошествии некоторого интервала времени, на линиях данных D0–D7 шины выставляется байт данных, который должен быть зафиксирован устройством после перехода строба записи в высокий уровень. Далее, процессор снимает адрес с линии адреса, снимает строб адреса и данных, заканчивая, таким образом, цикл записи данных в устройство.

Аппаратная реализация записи данных на стороне устройства не очень сложна. Функционально устройство ввода-вывода состоит из нескольких портов, которые схемотехнически представляют собой цифровые регистры с фиксацией данных (стробируемые регистры) для вывода данных и буферные усилители без стробирования для ввода данных в систему.

Устройство вывода с одним портом вывода применительно к временной диаграмме шины из рис. 2.3 можно представить приблизительно такой функциональной схемой (рис. 2.4):

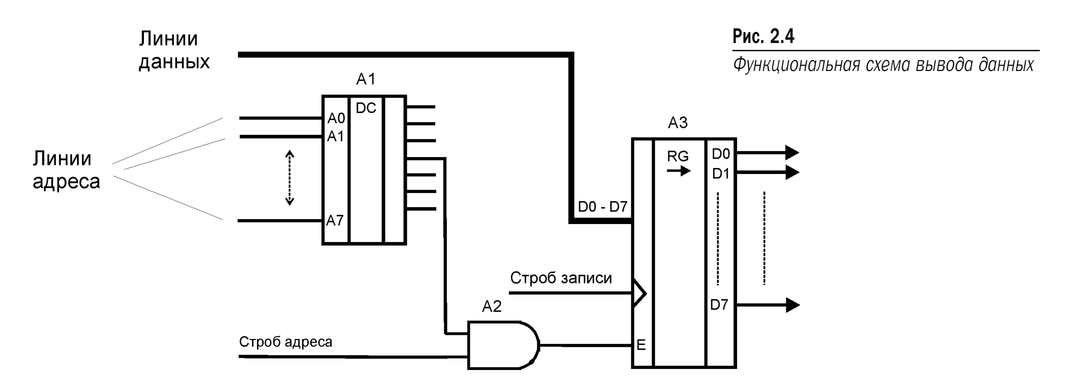


Рис. 2.4
Функциональная схема вывода данных

Здесь показана весьма упрощенная функциональная схема порта вывода данных. Если на шину адреса выставлен адрес порта вывода, то при высоком уровне строба адреса сигнал с выхода элемента «И» на микросхеме A2 разрешает работу регистра защелки A3. Если на шину данных будут выставлены данные (здесь показана 8-разрядная шина данных), то при перепаде строба записи из низкого в высокий уровень данные будут записаны в регистр A3. По этой схеме предполагается, что системная шина имеет отдельные линии адреса и данных, а адрес удерживается действительным во время всего цикла записи данных. Может возникнуть вопрос: нельзя ли вместо регистра-защелки использовать обычный буфер?

Такой вариант также возможен, но если записанные данные далее должны каким-то образом обрабатываться в устройстве, то следует их зафиксировать во избежание потери, что и выполняет регистр-защелка A3.

Чтение (ввод) данных с устройства может быть реализовано в простейшем цикле чтения нашей шины по схожей временной диаграмме, только вместо строба записи должен использоваться сигнал синхронизации чтения (строб чтения данных) (рис. 2.5).





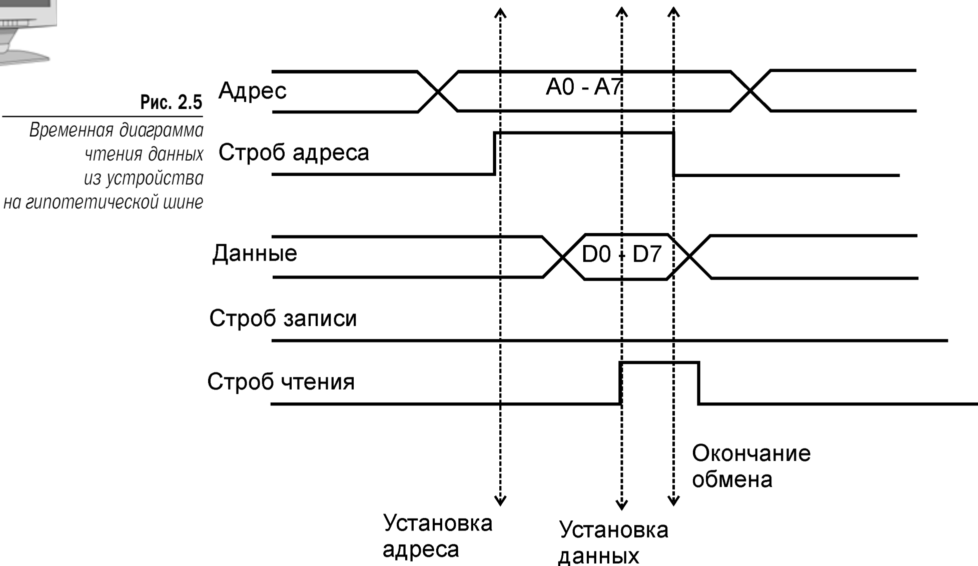


Рис. 2.5

Временная диаграмма чтения данных из устройства на гипотетической шине

В отличие от записи данных, сигнал строба записи теперь будет иметь низкий уровень, а чтение байта данных осуществляться по высокому уровню строба чтения.

Схемотехническая реализация устройства ввода данных не намного сложнее, чем при выводе данных. Функциональная схема порта ввода данных показана на рис. 2.6.

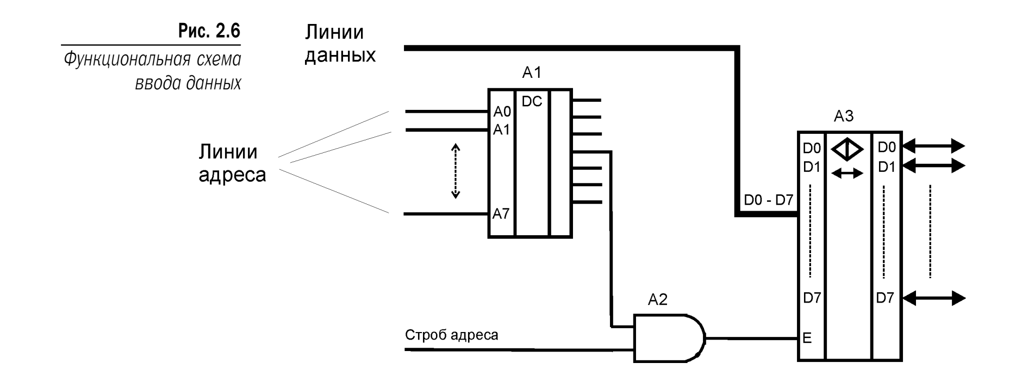


Рис. 2.6

Функциональная схема ввода данных

Порт ввода данных схемотехнически реализуется, как правило, проще, поскольку данные могут быть сразу прочитаны, без фиксации в регистре-защелке.

В схеме, показанной на рис. 2.6, для ввода данных с порта ввода используется тристабильный буфер A3. Опять-таки, для мультиплексированных шин данных типа PCI функциональная схема порта ввода будет сложнее. Комбинируя регистры ввода-вывода в одном устройстве, можно создать простое устройство ввода-вывода с двумя портами ввода-вывода. Приблизительно такая реализация шины ввода-вывода применяется в интерфейсе параллельного порта персонального компьютера, хотя количество портов ввода-вывода там больше (два порта вывода и один ввода).



Интерфейсы современных шин, естественно, намного сложнее, чем наша гипотетическая шина, что обусловлено влиянием нескольких факторов. Одним из них является размерность адресного пространства. В современных компьютерах используется как минимум 32-разрядная шина адреса, что потребовало бы как минимум 32 отдельных линий шины. Если учесть, что в компьютерных системах используется 8 бит данных, то только для линий адрес/данных потребуется как минимум 40 линий. Схемотехнически это выполнить очень сложно, особенно учитывая специфику работы с цепями сигналов высокой частоты. Для реализации эффективного обмена данными в этом случае в большинстве современных компьютерных шин используется механизм мультиплексирования, когда одни и те же линии данных используются в качестве линий адреса и данных.

Вторым важным моментом является то, что разные устройства имеют разное быстродействие, и система должна знать, готово ли устройство принять/передать данные, завершилась операция или нет и т. д. Реализовать обратную связь с устройством помогает механизм квитирования («рукопожатия»), когда начало любой операции на шине зависит от ответа устройства.

Эти механизмы реализованы в большинстве шин обмена данными, например, в той же PCI. Проанализируем, как происходит, например, запись данных в устройство на шине PCI (рис. 2.7).

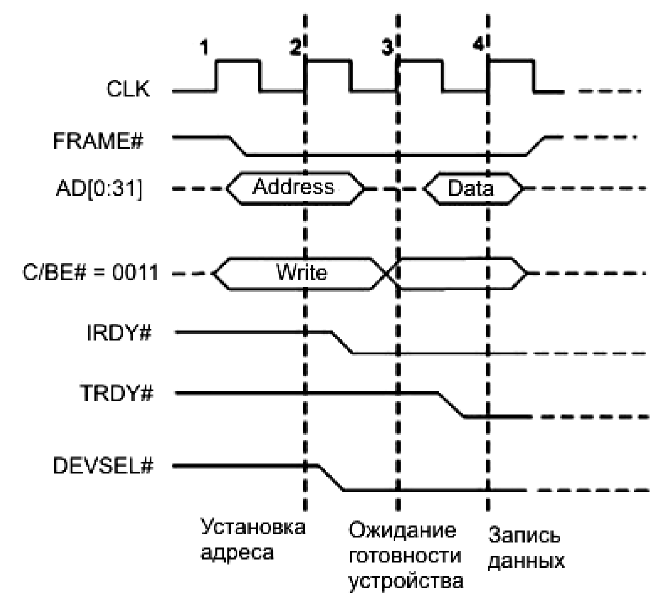


Рис. 2.7
Временная диаграмма работы цикла записи на шине PCI

Здесь я не буду анализировать назначение сигнальных линий PCI-шины – их назначение описано в многочисленной литературе и спецификациях на стандарт. По ходу я объясню смысл сигналов, присутствующих на шине. Самое первое, что следует отметить при анализе работы шины PCI – синхронизацию любых операций по фронту импульсов тактовой частоты CLK. Информация на шине будет действительна при перепаде сигнала CLK из высокого уровня в низкий, причем это касается команд, адресов, данных и сигналов состояния.

Новый цикл шины начинается с установки сигнала FRAME# в низкий уровень, который свидетельствует о выполнении операции на шине. Высокий уровень на этой линии запрещает обмен данными. После установки сигнала FRAME# в низкий уровень на линии адреса





AD[0:31] выставляется адрес устройства, которому необходимо передать данные. Одновременно на линии команд C/BE# выставляется код команды, которая будет выполняться (в данном случае, код равен 0011, что соответствует записи).

На линию DEVSEL# выставляется сигнал выбора устройства (активный уровень низкий). Наконец, контроллер шины выставляет на линию сигнал IRDY# готовности данных для записи устройства. Пока целевое устройство (target) не выставит сигнал готовности к приему данных на линию TRDY#, никакой записи данных в устройство не произойдет, и логика шины будет ожидать, пока сигнал на линии TRDY# не станет активным (низкий уровень).

Из временной диаграммы видно, что при 3-м тактовом импульсе CLK запись данных все еще невозможна, поскольку нет готовности устройства к приему данных (об этом свидетельствует высокий уровень на линии TRDY#). Наконец, устройство сигнализирует о готовности принять данные, которые и записываются в устройство по фронту 4-го синхроимпульса CLK. Цикл записи заканчивается при установке сигнала FRAME# в высокий уровень.

Как видно из описания цикла работы шины PCI, она, по существу, является асинхронной двунаправленной шиной обмена данными, скорость обмена на которой может варьироваться в зависимости от характеристик производительности устройства ввода-вывода. Аппаратный интерфейс устройства ввода-вывода для такой шины будет, естественно, несколько сложнее, чем тот, который мы рассмотрели для гипотетической шины обмена данными ранее в этой главе.

Что же касается программной части, то доступ к портам ввода-вывода осуществляется посредством операций чтения-записи с использованием стандартных инструкций процессора (для процессоров, совместимых с Intel, это команды in/out и их модификации). Если же устройство работает с регистрами, отображаемыми на память, то доступ к ним осуществляется с использованием инструкций mov. Данные, с которыми оперирует порт, могут иметь размер 1 байт, 2 байта (слово) и 4 байта (двойное слово).

Проанализируем, как выполнить чтение регистра состояния параллельного порта. Для этой цели можно использовать инструкцию in, например:

```
mov DX, 0x379
in AL, DX
```

Первая команда помещает в регистр DX адрес порта, а вторая читает данные размером в 1 байт в регистр AL процессора.

Из всего пространства адресов ввода-вывода (0–0xFFFF) часть адресов резервируется системой и не должна использоваться в программах пользователей — это адреса с 0xF8 по 0xFF.

Все устройства персонального компьютера взаимодействуют с процессором посредством шины PCI или PCI Express. При этом взаимодействие различных интерфейсов осуществляется с помощью специальных контроллеров, преобразующих сигналы на специализированных шинах в сигналы шин PCI или PCI Express. Например, для взаимодействия устройства USB с системой необходимо преобразовать сигналы на шине USB в сигналы шины PCI, для чего в компьютерной системе имеется хост-контроллер USB (как правило, он интегрируется в материнскую плату, хотя может быть выполнен и как отдельное устройство).

Второй пример. Для преобразования сигналов интерфейса RS-232, используемого при обмене данных через последовательный порт, в сигналы PCI на материнской плате имеется контроллер UART (асинхронного приемопередатчика). Специализированные контроллеры используются и для любых других интерфейсов(параллельного порта, Bluetooth, Wi-Fi и т. д.).



В более ранних операционных системах Windows 98/Me программы пользователя могли напрямую обращаться к портам ввода-вывода посредством инструкций процессора in и out, о которых я упоминал ранее. Для операционных систем Windows 2000/XP/2003, в которых код пользовательской программы не имеет доступа к адресному пространству системы (к которому принадлежат и порты ввода-вывода), выполнение операций с устройствами ввода-вывода возможно только посредством драйверов устройств, а прямой доступ к портам в некоторых случаях невозможен даже из драйвера.

При работе процессора в защищенном режиме защита доступа к портам ввода-вывода обеспечивается двумя механизмами:

- установкой уровня привилегий ввода-вывода в поле IOPL (I/O Provilege level) (биты 12-13 в регистре флагов процессора EFLAGS);
- картой, содержащей биты доступа в сегменте состояния задачи TSS (Task State Segment).

Поле IOPL в регистре флагов позволяет управлять доступом к пространству адресов посредством ограничения использования определенных инструкций процессора. Этот механизм защиты позволяет устанавливать необходимый для операционной системы уровень привилегий для выполнения операций ввода-вывода. В общем случае, доступ к адресному пространству ввода-вывода ограничен уровнями 0 и 1. В этом случае операции ввода-вывода могут выполнять драйверы ядра и драйверы устройств, в то время как приложения с меньшим уровнем привилегий операции ввода-вывода выполнить не смогут. Для менее привилегированных программ доступны механизмы ввода-вывода посредством вызовов функций интерфейса WinAPI.

Следующие инструкции будут выполнены только в том случае, если уровень привилегий выполняющейся программы (CPL, Current Privilege Level) меньше или равен установленному в IOPL: in, ins, out, outs, cli, sti.

Любая попытка выполнить эти инструкции в программе, уровень привилегий которой недостаточен, вызывает исключение общей защиты (General-Protection Exceptin). Поскольку каждая выполняющаяся задача получает собственную копию регистра флагов, то для нее устанавливается свой IOPL.

Рассмотрим инструкции процессора, используемые при выполнении операций ввода-вывода. Всех их можно условно разделить на две группы: инструкции, выполняющие пересылку одиночных данных (байта, слова или двойного слова), и инструкции, выполняющие пересылку строк байтов, строк слов и строк двойных слов.

Инструкции in (ввод данных с порта) и out (вывод данных в порт) передают данные между портом и регистрами EAX (32-разрядная операция), AX (16-разрядная операция) или AL (операция с одним байтом). Адрес порта может быть задан непосредственно в команде или посредством регистра DX.

Например,

```
in  AL, 0x378
mov DX, 0x379
out DX, AL
```

Строковые инструкции ins и outs пересылают данные между памятью и портом ввода-вывода. При этом адрес порта заносится в регистр DX, а адрес области памяти определяется одной из пар регистров DS:ESI (при вводе) или ES:EDI (при выводе данных).





При использовании префиксов повторения (например, rep) инструкции ins и outs выполняют поблочную передачу данных. В этом случае регистр ESI или EDI инкрементируется или декрементируется (в зависимости от значения флага DF), чтобы указывать на следующий байт, слово или двойное слово.

Следующие, более высокие уровни взаимодействия с устройствами используются операционными системами (системный уровень) и пользовательскими приложениями. Во всех современных операционных системах все устройства ввода-вывода, а также все файлы и каталоги для пользовательских процессов представлены как объекты файловой системы, а это означает, что доступ к объектам и операции с ними выполняются по некоторым унифицированным правилам, независимо от природы объекта (устройство, файл, именованный канал, сокет и т. д.). Такая модель позволяет пользовательским и системным процессам использовать для работы с объектами файловой системы набор стандартных функций интерфейса API. Замечу, что эта модель взаимодействия изначально была использована в операционных системах UNIX и хорошо себя зарекомендовала.

Программный код, выполняющийся в таких системах, может функционировать в одном из двух режимов: пользовательском или в режиме ядра. Большая часть программных компонентов операционной системы (системные службы, драйверы ядра и устройств и т. д.) функционирует в режиме ядра. Программный код режима ядра имеет практически неограниченный доступ к аппаратно-программным ресурсам системы, включая устройства ввода-вывода.

В рассмотренном только что примере чтение порта ввода с адресом 0x379 возможно только из программы, выполняющейся в режиме ядра. Например, программный код пользовательской программы в Windows работает только в режиме пользователя и взаимодействует с операционной системой посредством интерфейса прикладного программирования WinAPI, который реализован в виде подсистемы Win32.

В следующих главах на практических примерах мы увидим, как работать с такими устройствами ввода-вывода как параллельный и последовательный порты, звуковая карта, устройства USB и Bluetooth.

Параллельный порт в лабораторных разработках

3 Параллельный порт в лабораторных разработках

Параллельный порт персонального компьютера является одним из ранних аппаратно-программных интерфейсов, изначально разработанных для обмена данными с печатающими устройствами (принтерами). В настоящее время параллельный порт, помимо своего прямого назначения, довольно часто используется в разнообразных проектах для управления несложными аппаратными устройствами, а также для обработки и сбора данных.

Параллельный порт позволяет принимать 9 бит данных от устройства или отправлять 12 бит данных устройству. Аппаратно интерфейс реализован посредством 4-х линий управления, 5-ти линий состояния и 8-ми линий данных, которые подключены к 25-контактному разъему типа DB-25, расположенному на задней стенке компьютера.

Самым первым стандартом параллельного порта стал интерфейс, известный под названием Centronics. Он описывает сигналы, протокол обмена и расположение внешних контактов на разъемах в компьютере и принтере. С точки зрения схемотехники, интерфейс Centronics реализован как группа из трех регистров ввода-вывода: данных, управления и состояния, которые доступны программисту.

В таблице 3.1 приведено описание сигналов интерфейса, нумерация выводов на разъеме DB-25 в компьютере и направление сигналов по отношению к параллельному порту компьютера.

Указанная спецификация сигналов, соответствующая интерфейсу Centronics, была стандартизована под названием SPP (Standard Parallel Port). В дальнейшем спецификации режимов обмена данными получили дальнейшее развитие в протоколах EPP (Enhanced Parallel Port) и ECP (Extended Capabilities Port), которые были разработаны и приняты под эгидой стандарта IEEE 1284, обеспечивая обратную совместимость со стандартным режимом SPP.

Проанализируем упрощенную временную диаграмму обмена для параллельного порта, работающего в стандартном режиме SPP или, проще говоря, Centronics. Перед анализом работы интерфейса следует отметить, что при обмене данными между устройствами одно из них является инициатором (источником) обмена, а другое — приемником.

Для стандартного интерфейса SPP источником обмена является обычно компьютер, или, по–другому, хост, а приемником — принтер или иное устройство. Временная диаграмма обмена по протоколу SPP показана на рис. 3.1.

Рассмотрим пример передачи байта данных от источника к приемнику. Передача выполняется в следующей последовательности:

- вначале проверяется состояние сигнала BUSY: если он равен 0, то источник приступает к передаче байта данных, выставляя на линиях D0–D7 биты данных, после чего



Таблица 3.1

Спецификация интерфейса Centronics

Номер контакта	Сигнал интерфейса	Описание сигнала	Бит регистра	Направление сигнала	Активный уровень сигнала
1	Strobe	Строб готовности данных	Управления–0	Ввод–вывод	Низкий, инверсия
2	Data0	Бит данных 0		Вывод	
3	Data1	Бит данных 1		Вывод	
4	Data2	Бит данных 2		Вывод	
5	Data3	Бит данных 3		Вывод	
6	Data4	Бит данных 4		Вывод	
7	Data5	Бит данных 5		Вывод	
8	Data6	Бит данных 6		Вывод	
9	Data7	Бит данных 7		Вывод	
10	Ack	Подтверждение приема данных	Состояния–6	Ввод	
11	Busy	Готовность приема данных	Состояния–7	Ввод	Инверсия
12	Paper-Out Paper-End	Конец бумаги	Состояния–5	Ввод	
13	Select	Принтер включен/выключен	Состояния–4	Ввод	
14	AutoLineFeed	Автоматический перевод строки	Управления–1	Ввод–вывод	Инверсия
15	Error/Fault	Ошибка устройства	Состояния–3	Ввод	
16	Initialize	Инициализация устройства	Управления–2	Ввод–вывод	
17	Select-Printer Select-In	Выбор принтера	Управления–3	Ввод–вывод	Инверсия
18–25	Gnd	Общий			

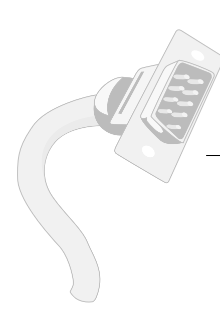


Рис. 3.1

Обмен данными по протоколу SPP

устанавливает сигнал STROBE в низкий уровень, указывая на то, что данные на линиях D0–D7 действительны;

- приемник читает байт с шины данных D0–D7, устанавливая сигнал BUSY в высокий уровень. По окончании обработки данных приемник устанавливает сигнал ACK в низкий уровень, подтверждая окончание обработки, и снимает сигнал BUSY. Если в течение определенного интервала времени (как правило, это 5 микросекунд) сигнал ACK не будет установлен, то источник полагает, что произошла ошибка тайм-аута со стороны приемника. В большинстве промышленных разработок этот сигнал в целях экономии времени игнорируется.



Кроме того, если после приема очередного байта приемник пока не готов принимать следующий байт данных, то он оставляет сигнал BUSY в активном состоянии.

Изначально параллельный порт персонального компьютера был реализован как отдельное устройство, но в настоящее время схемотехника порта интегрирована в материнскую плату компьютера. Напомню, что базовая программная модель параллельного порта включает три регистра: данных, состояния и управления.

Базовым адресом параллельного порта является адрес регистра данных, по отношению к которому вычисляются остальные адреса.

Например, если базовый адрес порта равен некоторому значению Base, то регистр данных будет иметь адрес Base, регистр состояния — Base+1 и регистр управления — Base+2.

В качестве базовых адресов регистров чаще всего используются адреса (в шестнадцатеричной нотации) 0x3BC, 0x378, 0x278, которые обычно прошиваются в BIOS материнской платы. Эти адреса, как и режим работы порта, можно изменить, установив соответствующие настройки в BIOS.

Драйвер параллельного порта для обработки данных может использовать аппаратное прерывание; как правило, это линия IRQ7 или IRQ9 контроллера прерываний. Базовый адрес 0x3BC использовался в более ранних моделях комбинированных видеокарт, а в настоящее время нередко используется как вариант выбора для интегрированного на материнской плате параллельного порта. В операционных системах Windows и Linux параллельным портам присваиваются символические имена в виде «LPTn», где n — номер порта (1, 2 и т. д.), причем «LPT1» обычно присваивается порту с базовым адресом 0x378, а «LPT2» — порту с базовым адресом 0x278h, как показано в таблице 3.2.

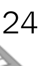
Таблица 3.2
Базовые адреса параллельного порта

Адреса	Примечание
0x3BC —0x3BE	Не поддерживается в ECP
0x378 —0x37F	Порт LPT1
0x278 —0x27F	Порт LPT2

При включении компьютера BIOS изначально выполняет проверку по адресу 0x3BC, и если обнаруживает, что он принадлежит параллельному порту, то присваивает порту имя «LPT1». Далее выполняется проверка адресов 0x378 и 0x278; и если адреса обнаружены, им ставятся соответственно имена «LPT2» и «LPT3». Поскольку в большинстве современных материнских плат адрес 0x3BC для параллельного порта не используется, то базовым адресом «LPT1» устанавливается 0x378, а базовым адресом «LPT2» — 0x278.

Параллельные порты современных компьютеров могут работать в разных режимах, поэтому в большинстве версий BIOS предусмотрена установка следующих режимов:

- Printer (иногда этот режим называется Default или Normal);
- Standard & Bi-directional (SPP);
- EPP1.7 и SPP;
- EPP1.9 и SPP;
- ECP;
- ECP и EPP1.7;
- ECP и EPP1.9.



Приведем их краткое описание:

- Printer — это название базового режима, соответствующего стандарту SPP. В данном режиме параллельный порт работает только как однонаправленный, поэтому бит 5-го регистра управления в этой конфигурации не задействован;
- Standard & Bidirectional — порт работает как двунаправленный. В этом режиме бит 5-го регистра управления позволяет устанавливать двусторонний обмен данными, что позволяет источнику прочитать биты на линиях данных;
- EPP1.7 и SPP — этот режим представляет собой комбинацию режимов EPP 1.7 и SPP. В этом случае доступны как стандартные регистры режима SPP (данных, состояния и управления), так и EPP-регистры. Направление передачи сигналов порта можно менять, используя бит 5-го регистра управления. Стандарт EPP 1.7 является более ранней версией EPP, поэтому в нем не предусмотрен бит тайм-аута;
- EPP1.9 and SPP — этот режим напоминает предыдущий, с той лишь разницей, что используется версия EPP 1.9.

Интерфейс Centronics благодаря его простоте можно успешно использовать в собственных разработках устройств ввода-вывода или лабораторных системах управления оборудованием.

Следует отметить, что прямое программирование портов ввода-вывода посредством команд ассемблера возможно только в операционных системах Windows 98/Me, но невозможно в защищенных системах Windows 2000/XP/2003. Тем не менее, даже в таких системах можно получить доступ к аппаратным ресурсам параллельного порта, если использовать свободно распространяемый драйвер PortTalk (один из сайтов, откуда его можно скачать, www.beyondlogic.org). Этот драйвер несложен в применении, и мы будем его применять при разработке проектов этой главы.

При разработке собственных проектов с использованием параллельного порта вам необходимо учитывать следующие факторы:

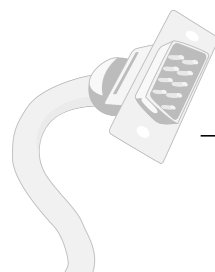
- адреса базовых регистров параллельного порта на вашей машине могут отличаться от используемых в примерах этой главы, поэтому нужно их дополнительно проверить и установить корректные значения в исходном тексте программы;
- выходы параллельного порта являются слаботочными, допускающими подключение нагрузки с потреблением тока всего в несколько миллиампер, поэтому непосредственно к ним нельзя подключать электрические цепи, потребляющие значительный ток!

Во всех примерах данной главы базовым адресом параллельного порта является стандартный по умолчанию адрес в BIOS, равный 0x378, поэтому если вы используете в проектах другой адрес, то внесите соответствующие изменения в тексты программ, где это необходимо!

3.1. Организация ввода-вывода данных через параллельный порт

Общеизвестно, что параллельный порт используется для подключения устройств печати, хотя этим его применение не исчерпывается — интерфейс параллельного порта является очень удобным для создания различных аппаратно-программных модулей управления элек-





тронными исполнительными устройствами и устройств обработки данных как в промышленности, так и лабораторных исследованиях.

Известно, что современные операционные системы Windows 2000/XP/2003 не позволяют пользовательским программам напрямую работать с регистрами параллельного порта (как, впрочем, и с другими портами в адресном пространстве ввода-вывода), поскольку доступ к портам ввода-вывода могут получить только процессы, выполняющиеся в режиме ядра, например, драйверы устройств и некоторые системные сервисы.

В более ранних операционных системах линейки Windows 95/98/Millenium такой проблемы не было, поскольку все ресурсы системы были доступны пользовательскому процессу, и управлять вводом-выводом через порт можно было непосредственно инструкциями ассемблера `in` и `out`. Реализовать ввод-вывод данных в параллельный порт в Windows можно, используя функции Win API `CreateFile`, `ReadFile` и `WriteFile`, хотя это будет несколько сложнее и потребует разработки более сложного аппаратного интерфейса управляемого устройства.

Для управления параллельным портом посредством функций WINAPI необходимо вначале открыть устройство функцией `CreateFile`, задав в качестве первого параметра «LPTn», где n — номер одного из параллельных портов, присутствующих в системе. Обычно n равно 1, т. е. используется «LPT1». После получения дескриптора устройства с помощью функции `CreateFile` можно выполнять операции чтения-записи в устройство. Пример доступа к параллельному порту посредством функций WINAPI показан далее:

```
#include <windows.h>
#include <stdio.h>
. . .
HANDLE hLPT;
char *buf = "Any string";
DWORD bytes;
. . .
hLPT = CreateFile("LPT1",
                   GENERIC_READ | GENERIC_WRITE,
                   0,
                   NULL,
                   OPEN_EXISTING,
                   0,
                   NULL);
if (hLPT == INVALID_HANDLE_VALUE)
{
  printf("Could not open file (error %d)\n", GetLastError());
  return 0;
}
bool fSuccess = WriteFile(hLPT,
                          buf,
                          strlen(buf),
                          &bytes,
                          NULL);
. . .
```

Таким образом можно вывести данные на принтер в программе, но управлять записью данных непосредственно в регистр устройства, разработанного пользователем, не получится. Указанный выше фрагмент кода работает с устройством посредством драйвера операционной системы.

Для ввода-вывода данных в параллельный порт в операционных системах Windows 2000/XP/2003/Vista мы будем использовать драйвер PortTalk, позволяющий осуществлять доступ к физическим портам ввода-вывода напрямую. Принцип работы драйвера мы не будем рассматривать подробно, скажу лишь, что его основная функция – очистить бит защиты, который устанавливается для всех приложений Windows, работающих в пользовательском режиме. Данный бит запрещает приложению, работающему в пользовательском режиме, обращаться напрямую к защищенным ресурсам системы, к которым относятся регистры ввода-вывода физических устройств и системные области памяти. Если бит сбросить, то ваше приложение сможет получить доступ к этим ресурсам, причем программировать порты ввода-вывода можно будет с помощью языка ассемблера точно так же, как это имеет место в Windows 98/Me. Перейдем к практическим аспектам работы с параллельным портом.

3.2. Интерфейсы ввода-вывода дискретных сигналов параллельного порта

Вначале остановимся на методике вывода данных в параллельный порт. Вывод данных в параллельный порт можно использовать для управления различными устройствами, для генерации импульсных последовательностей и т. д. Регистры параллельного порта работают с уровнями TTL-логики, поэтому их легко сопрягать с цифровыми микросхемами.

Обычно для вывода используется регистр данных порта с шестнадцатеричным адресом 0x378 (стандартное значение) или 0x278 (обычно используется для LPT2).

Разработаем нашу первую программу для работы с параллельным портом, для чего запустим среду Delphi 2007 (можно также работать в одной из сред Delphi 7, Delphi 2005 или Turbo Delphi) и создадим графическое приложение, в котором при нажатии кнопки, размещенной на форме, выходной код регистра 0x378 будет инвертировать свое содержимое. Вот как выглядит поле конструктора приложения (рис. 3.2):

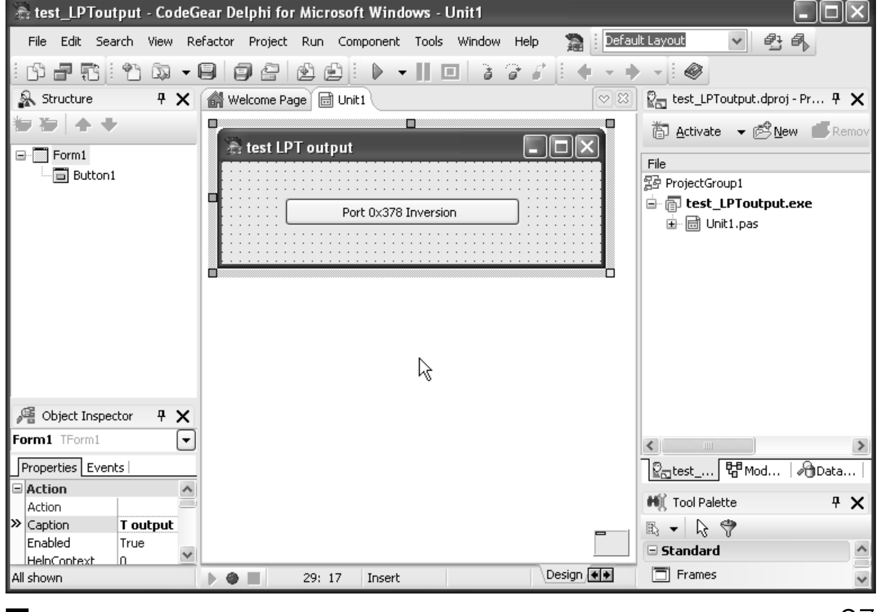


Рис. 3.2
Вид окна конструктора приложения

Исходный текст приложения выглядит так, как показано далее:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
 asm
   mov  DX, 378h
   in   AL, DX
   not  AL
   out  DX, AL
 end;
end;

end.
```

Основные действия выполняются в блоке `asm … end`, который содержит группу ассемблерных команд. В регистр `DX` заносится адрес порта, затем в регистр `AL` командой `in AL, DX` считывается содержимое порта, которое инвертируется и выводится обратно в регистр порта.

Для запуска откомпилированной программы необходимо в каталог программы скопировать файлы porttalk.sys, allowio.exe и start.bat. Командный файл start.bat должен включать следующие строки:

```
@echo off
AllowIo 0x378 test_LPToutput.exe
```



Программа allowio.exe снимает бит защиты с портов, указанных в качестве первых аргументов, а последним аргументом должно быть путевое имя файла исполняемой программы. Результат работы программы легко проверить с помощью тестера, подсоединив общий провод устройства к контакту 25 разъема LPT на задней стенке системного блока, а второй к какому-либо контакту, начиная с номера 2 и заканчивая 9-м. При нажатии на кнопку в окне работающего приложения будут меняться и показания тестера.

Здесь необходимо отметить следующее. При считывании регистра данных параллельного порта мы считываем фактически значение триггеров-защелок, а не значения на контактах разъема. Для считывания данных с внешних контактов (например, контакт датчика) следует использовать регистр состояния с шестнадцатеричным адресом 0x379. Входы этого регистра можно использовать для контроля каких-либо дискретных сигналов от цифровых схем.

Пусть, например, нужно считать сигнал с фотодатчика, реагирующего на дневной свет. В этом случае для контроля освещенности можно использовать схему устройства, представленную ниже (рис. 3.3).

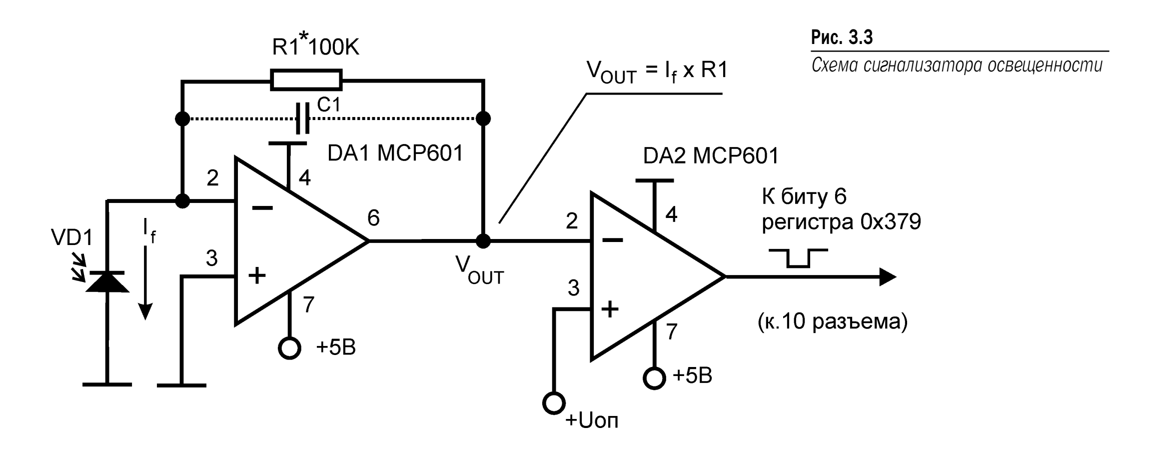


Рис. 3.3
Схема сигнализатора освещенности

В этой схеме при попадании светового потока на фотодиод D1 в нем генерируется ток I_f, который проходя через резистор R1, включенный в цепь отрицательной обратной связи операционного усилителя DA1, создает на его выходе напряжение $V_{OUT} = I_f \times R1$. Чем мощнее будет световой поток, тем больше будет значение тока и, соответственно, выше напряжение V_{OUT} (естественно, оно должно попадать в диапазон допустимых входных напряжений для данного усилителя при указанном напряжении источника питания). Конденсатор C1 служит для подавления возможной генерации при возникновении положительной обратной связи, хотя может и отсутствовать, если схема работает устойчиво. Его емкость подбирается экспериментально и может составлять от нескольких сотен до нескольких тысяч пикофарад, в зависимости от номинала резистора R1.

Выходное напряжение операционного усилителя DA1 поступает на вход компаратора напряжения DA2, где сравнивается с опорным напряжением U_{OP}. При превышении V_{OUT} над U_{OP} на выходе компаратора появляется напряжение низкого уровня, которое приходит на 6-й разряд регистра состояния параллельного порта (адрес 0x379). Далее этот бит может быть проанализирован программным обеспечением и выполнены какие-либо действия (в нашей программе, которую мы далее рассмотрим, просто будет выведено сообщение на экран дисплея).

Разработаем простое графическое приложение для схемы контроля освещенности. Вот как выглядит окно конструктора в среде Delphi 2007 (рис. 3.4).

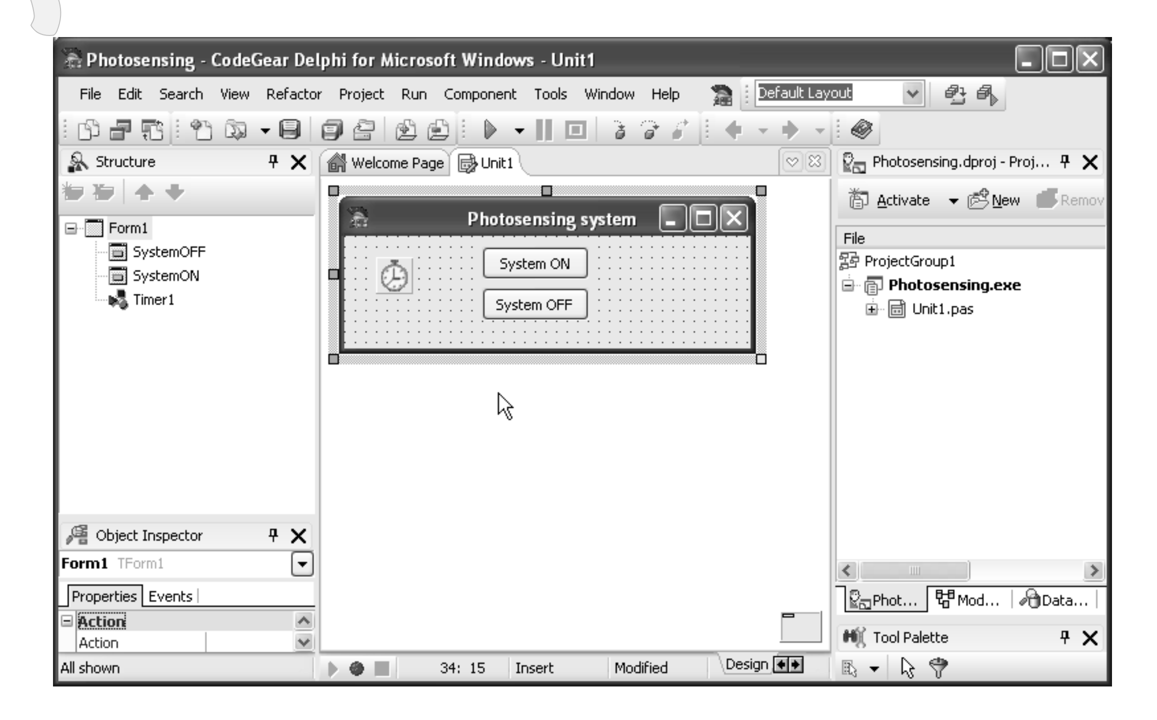


Рис. 3.4
Окно конструктора приложения

На форме приложения разместим три визуальных компонента: две кнопки и таймер. Таймер выполняет здесь основную работу – он считывает каждые 5 секунд значение бита 6 порта 0x379 и, если оно равно 0 (датчик сработал), то инициирует вывод сообщения на экран дисплея. Кнопка **SystemON** запускает, а кнопка **SystemOFF** останавливает работу таймера. Как видите, программа довольно проста. Вот ее исходный текст:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    SystemON: TButton;
    Timer1: TTimer;
    SystemOFF: TButton;
    procedure SystemONClick(Sender: TObject);
```

```
    procedure SystemOFFClick(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.SystemONClick(Sender: TObject);
begin
 timer1.Enabled:= true;
end;

procedure TForm1.SystemOFFClick(Sender: TObject);
begin
 timer1.Enabled:= false;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
var
 LightON: Byte;
begin
 asm
  mov   DX, 379h
  in    AL, DX
  and   AL, 01000000b
  shr   AL, 6
  mov   byte ptr LightON, AL
 end;
 if LightOn = 0 then
   ShowMessage('Light is ON!');
end;

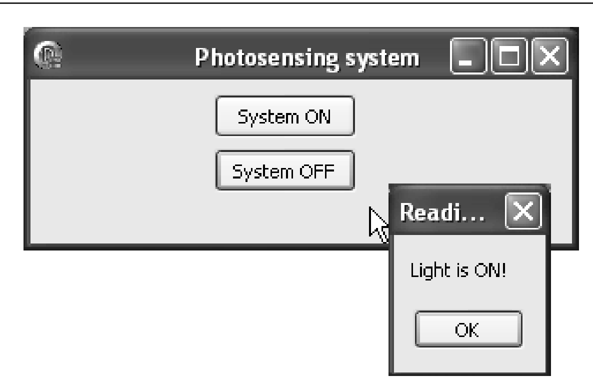
end.
```

Основные действия выполняются в блоке `asm ... end` функции-обработчика события таймера. Полученный из регистра 0x379 байт сохраняется в регистре `AL`. Затем содержимое регистра `AL` сдвигается вправо на 6 позиций, так, чтобы интересующий нас бит оказался на месте младшего бита. Затем значение `AL` помещается в переменную `LightON`, и если равно 0, то на экран выводится соответствующее сообщение с помощью оператора `ShowMessage`.

Вот так выглядит окно работающей программы (рис. 3.5).

Рис. 3.5

Окно работающей программы



Для запуска этой программы, как и в предыдущем примере, нам понадобятся файлы porttalk.sys, allowio.exe и start.bat, причем в командном файле start.bat необходимо указать имя программы.

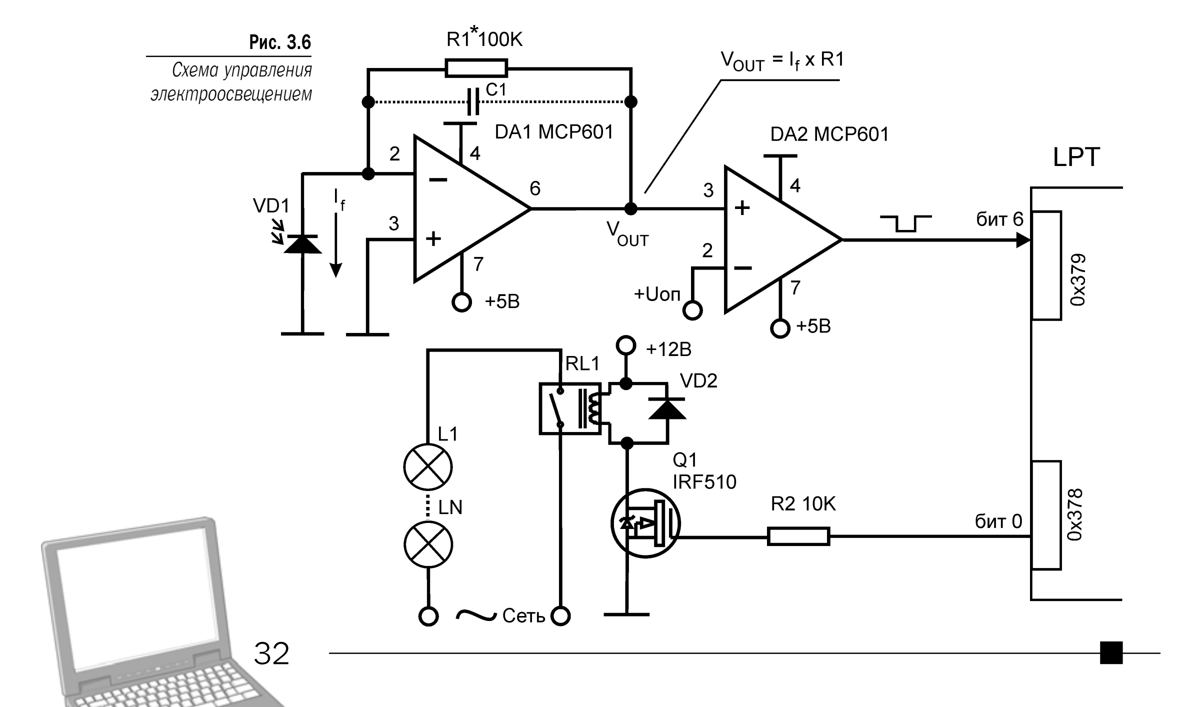
Этот проект, немного доработав, можно превратить в простую систему управления освещением в помещении. Такая система может работать следующим образом: при снижении уровня освещенности (например, с наступлением сумерек) ток фотодатчика уменьшался бы настолько, что вход 6 регистра состояния переключался бы в состояние 0, а программа, обнаружив это, включала бы освещение. Включение электроосвещения можно реализовать через один из разрядов регистра данных, например, нулевой.

Аппаратную часть такой системы управления освещением можно разработать на основе схемы из рис. 3.3 путем небольшой модернизации последней. Вот как могла бы выглядеть система управления освещением с управлением от параллельного порта (рис. 3.6).

Схема работает следующим образом: при уменьшении тока через фотодиод VD1 с уменьшением освещенности падает напряжение на выходе операционного усилителя DA1. Как

Рис. 3.6

Схема управления электроосвещением



только это напряжение станет меньше напряжения Uоп, компаратор переключается, и напряжение на его выходе становится равным лог.0. Программа, обнаружив нулевой уровень в 6-м разряде регистра состояния с адресом 0x379, устанавливает высокий уровень напряжения в 0-м разряде регистра данных с адресом 0x378. Это напряжение открывает мощный полевой транзистор Q1, что вызывает протекание тока через обмотку реле RL1. В результате контакты реле замыкают вторичную силовую цепь с подключенными осветительными приборами L_1 . . L_N. Обратно, если освещенность увеличивается, то обратный ток через фотодиод VD1 возрастает, что приводит к срабатыванию компаратора и переключению его выхода в состояние лог.1. Программа, прочитав бит 6, отключает освещение, установив уровень лог.0 в 0-м разряде регистра данных.

Тип реле и диода VD2 в этой схеме подбираются в зависимости от нагрузки во вторичной цепи и тока в обмотке, необходимого для срабатывания реле. На фотодиод не должен попадать свет от включенного освещения, иначе программа будет непрерывно переключать цепь осветителей.

Исходный текст программы, управляющей освещением, показан далее:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, StdCtrls, ExtCtrls;

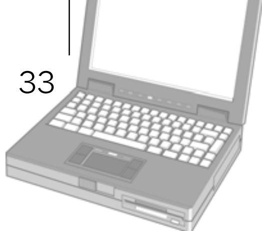
type
  TForm1 = class(TForm)
    SystemON: TButton;
    Timer1: TTimer;
    SystemOFF: TButton;
    procedure SystemONClick(Sender: TObject);
    procedure SystemOFFClick(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.SystemONClick(Sender: TObject);
begin
  timer1.Enabled:= true;
```



```
end;

procedure TForm1.SystemOFFClick(Sender: TObject);
begin
 timer1.Enabled:= false;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
 asm
  mov   DX, 379h
  in    AL, DX
  and   AL, 01000000b
  shr   AL, 6
  dec   DX
  cmp   AL, 0
  sete  BL
  mov   AL, BL
  out   DX, AL
 end;
end;

end.
```

Как и в программе из предыдущего примера, здесь используются те же визуальные компоненты (две кнопки и таймер), но исходный текст существенно изменен. Эти изменения коснулись программы-обработчика события таймера. Если при считывании 6–го бита регистра 0x379 он установлен в 0 (низкий уровень освещенности в помещении), то бит 0-го регистра 0x378 устанавливается в лог.1, открывая транзистор Q1, который включает цепь осветителей. Если значение 6–го бита равно лог.1 (хорошая освещенность), то бит 0-го регистра 0x378 сбрасывается в 0, выключая цепь освещения.

3.3. Интерфейсы аналоговых сигналов

Большинство измерений физических величин оперируют с аналоговыми, т. е. непрерывными сигналами, изменяющимися во времени. Для обработки таких сигналов в компьютере необходимо преобразовать их в цифровую форму. Этот процесс называется аналого-цифровым преобразованием, и он является своеобразным мостом между «аналоговым» физическим миром и цифровым миром компьютеров. Для преобразования используются специальные микросхемы аналого-цифровых преобразователей. Мы рассмотрим несколько простых реализаций аналого-цифровых преобразований, в которых данные будут приниматься через параллельный порт персонального компьютера, работающего под управлением операционных систем Windows.

Вначале разработаем проект простого преобразователя, работающего на микросхеме LTC1286 фирмы Linear Technology. Это устройство представляет собой 12-битовый преобразователь, управление преобразованием данных в котором осуществляется по трем сигнальным линиям (рис. 3.7):

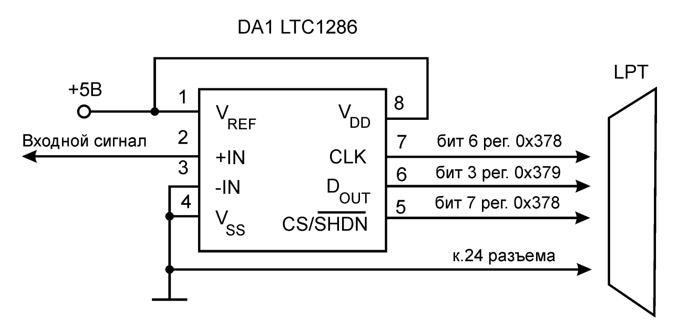


Рис. 3.7
Схема подключения аналого-цифрового преобразователя к параллельному порту

Микросхема LTC1286 представляет собой микромощный 12-разрядный аналого-цифровой преобразователь, работающий по методу последовательных приближений. LTC1286 позволяет выполнить до 12.5 тыс. преобразований в секунду, потребляя при этом ток около 250 микроампер. В режиме ожидания микросхема потребляет всего 1 наноампер. При таких характеристиках энергопотребления для питания устройства можно использовать батареи или аккумуляторы. Микросхема работает с питающими напряжениями от 5 до 9 вольт и может выполнять преобразование как однополярных, так и двухполярных входных сигналов.

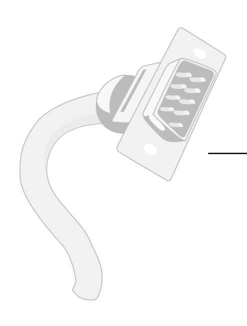
Результат преобразования передается в параллельный порт персонального компьютера по протоколу SPI. Как известно, протокол SPI требует наличия всего 3-х сигнальных линий, что значительно облегчает подключение преобразователя к удаленному оборудованию, например, к микроконтроллерам или удаленным системам управления. Рассмотрим принципиальную схему преобразователя.

В данной схеме устройство работает с однополярными входными напряжениями в диапазоне 0-5В.

Микросхема LTC1286 подключается к схеме следующим образом:

- `Vref` (`Выв.1`) — к этому выводу подключается источник опорного напряжения, определяющий диапазон преобразуемого напряжения. В данном случае вывод источника опорного напряжения подключен к источнику напряжения +5В, что и определяет диапазон преобразования от 0 до 5В;
- `+IN` (`Выв.2`) — к этому выводу подключается источник сигнала положительной полярности;
- `-IN` (`Выв.3`) — к этому выводу подключается источник сигнала отрицательной полярности (в данном случае вывод заземлен, поскольку преобразователь включен для обработки однополярного сигнала положительной полярности);
- `Vss` (`Выв.4`) — к этому выводу непосредственно подключается общий провод аналоговой части схемы («аналоговая земля»);
- `CS/SHDN` (`Выв.5`) — выбор кристалла (Chip Select, CS). Логический «0» на этом входе разрешает работу микросхемы LTC1286. Логическая «1» на этом входе запрещает работу микросхемы, переводя ее в режим минимального потребления мощности;
- `Dout` (`Выв.6`) — выход цифрового сигнала. На этом выводе появляется последовательность двоичных битов, являющаяся результатом преобразования;
- `CLK` (`Выв.7`) — вход тактовой частоты. На этот вход подаются импульсы тактовой частоты, синхронизирующие последовательность битов на выводе Dout. Кроме того, частота тактовых импульсов определяет скорость преобразования;
- `Vdd` (`Выв.8`) — к этому выводу подключается источник питания. Он должен обеспечивать по возможности минимальный уровень пульсаций, поскольку это сказывается





на точности преобразования. В данной конструкции используется широко распространенный стабилизатор типа 7805, а в качестве входного напряжения для стабилизатора служит обычная 9–вольтовая батарея или аккумулятор. Поскольку преобразователь потребляет незначительную мощность, то вполне оправдано применение батарейного питания, которое обеспечивает длительную работу преобразователя. Кроме того, при батарейном питании отсутствуют пульсации и помехи, свойственные сетевым источникам питания, что обеспечивает высокую точность преобразования.

Несколько слов о конструкции преобразователя. Для соединения устройства с параллельным портом ПК желательно использовать плоский кабель длиной не более 30 см. Если же необходим более длинный кабель, то следует применить буферные усилители/формирователи сигналов. Все соединения на плате преобразователя необходимо выполнить проводниками минимальной длины и использовать для питания источник с минимальными пульсациями, либо предусмотреть варианты фильтрации помех по шине питания.

Программное обеспечение аналого-цифрового преобразователя разработано, как и в предыдущих примерах, с использованием среды Delphi 2007.

Форма приложения с размещенными на ней компонентами показана на рис. 3.8.

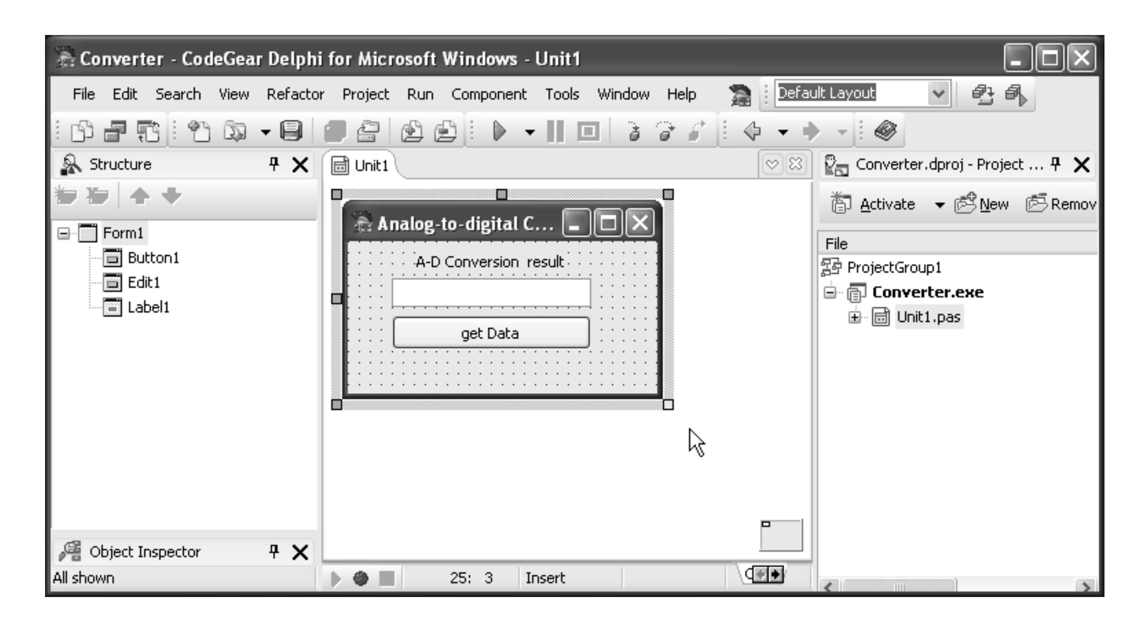


Рис. 3.8
Окно конструктора приложения

Программа работает очень просто: при нажатии кнопки **get Data** в поле однострочного редактора появляется результат измерений, как показано на рис. 3.9.

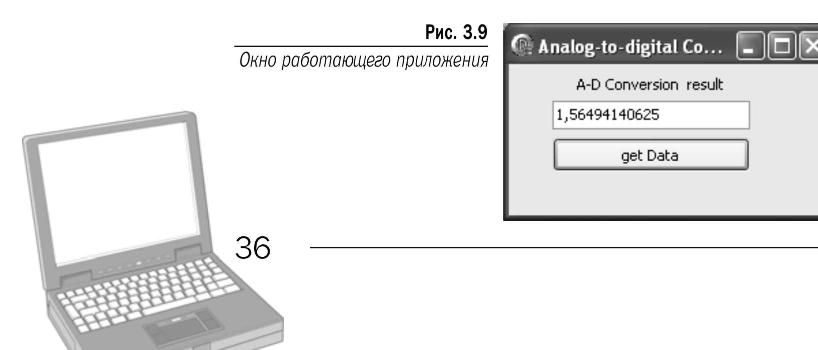


Рис. 3.9
Окно работающего приложения

Исходный текст приложения показан далее:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Button1: TButton;
    Label1: TLabel;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  binResult: Word;
  total: Real;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
const
 DATA = #$378;
 STATUS = #$379;
begin
 asm
   push ebx

   mov  dx, DATA
   xor  ax, ax
   bts  ax, 7
   out  dx, al

   btr  ax, 7
   out  dx, al

   mov  bx, 15
 @next:
```




```
   xor  ax, ax
   mov  dx, DATA
   btr  ax, 6
   out  dx, al


   mov  dx, STATUS
   in   al, dx
   bt   ax, 3
   rcl  cx, 1

   mov  dx, DATA
   bts  ax, 6
   out  dx, al

   dec  bx
   jnz  @next

   mov  dx, DATA
   bts  ax, 7
   out  dx, al

   pop  ebx
   and  cx, 0FFFh
   mov  word ptr binResult, cx
 end;
 total:= binResult*5.0 / 4096;
 Edit1.Text:= FloatToStr(total);
end;

end.
```

Наше приложение выводит результат измерений в окно однострочного редактора. Во многих случаях для обработки данных применяются офисные расширения, такие, например, как Microsoft Excel или Microsoft Access. Модифицируем наше приложение таким образом, чтобы можно было получать данные непосредственно в таблицу Excel.

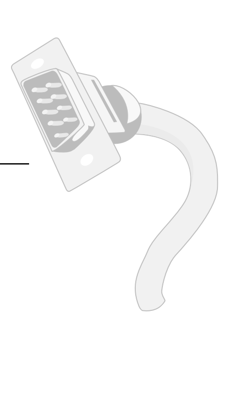
В этом случае потребуется внести некоторые изменения в исходный текст программы, чтобы можно было работать с Excel. Модифицированный листинг программы показан далее:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, StdCtrls, ComObj;

type
```




```
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  binResult: Word;
  total: Real;
  XLApp: variant;
  i1: Integer;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
const
 DATA = #$378;
 STATUS = #$379;
begin
 asm
   push ebx

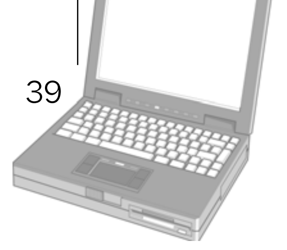
   mov  dx, DATA
   xor  ax, ax
   bts  ax, 7
   out  dx, al

   btr  ax, 7
   out  dx, al

   mov  bx, 15
 @next:

   xor  ax, ax
   mov  dx, DATA
   btr  ax, 6
   out  dx, al


   mov  dx, STATUS
   in   al, dx
   bt   ax, 3
```

```
    rcl  cx, 1

    mov  dx, DATA
    bts  ax, 6
    out  dx, al

    dec  bx
    jnz  @next

    mov  dx, DATA
    bts  ax, 7
    out  dx, al

    pop  ebx
    and  cx, 0FFFh
    mov  word ptr binResult, cx
  end;
  total:= binResult*5.0 / 4096;
  XLApp.ActiveSheet.cells.item[i1, 1].value:= total;
  Inc(i1);


end;

procedure TForm1.FormCreate(Sender: TObject);
begin
 XLApp := CreateOleObject("Excel.Application");
 XLApp.Workbooks.Add;
 XLApp.Visible := True;
 i1:= 1;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
 if not VarIsEmpty(XLApp) then
  begin
    XLApp.DisplayAlerts := False;  // Discard unsaved files....
    XLApp.Quit;
  end;
end;
end.
```

Первое, на что хочу обратить внимание читателей,– это то, как реализован вызов объекта Excel из приложения Delphi. При создании окна приложения (событие **FormCreate**) создается также и объект Microsoft Excel. Кроме того, по окончанию работы приложения (событие **FormDestroy**) объект Microsoft Excel должен быть уничтожен.

Вывод данных в таблицу Excel осуществляется при нажатии на кнопку **get Data**, при этом данные формируются в первом столбце таблицы.

Окно конструктора модифицированного приложения выглядит следующим образом (рис. 3.10).



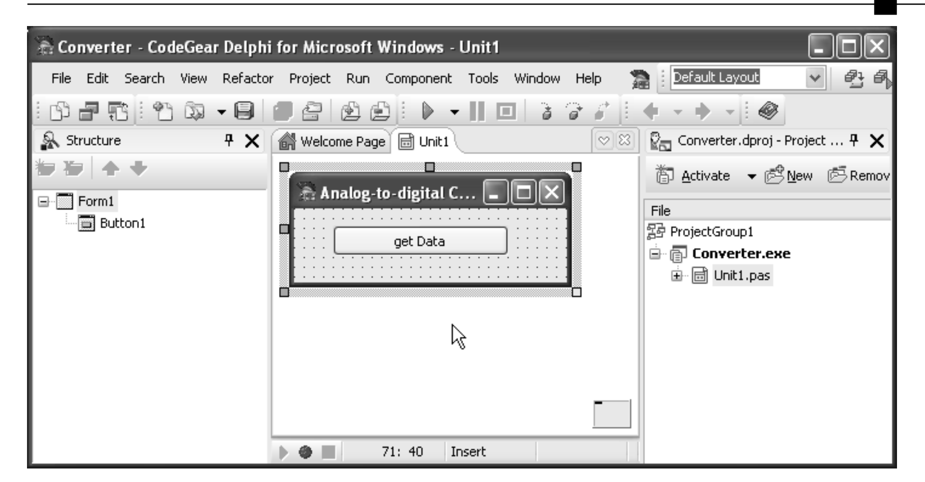


Рис. 3.10
Окно конструктора модифицированного приложения

При выполнении приложения на экране дисплея будут присутствовать два окна: окно приложения и окно таблицы Excel. Каждый раз при нажатии кнопки в окне приложения содержимое таблицы в первом слева столбце будет изменяться (рис. 3.11).

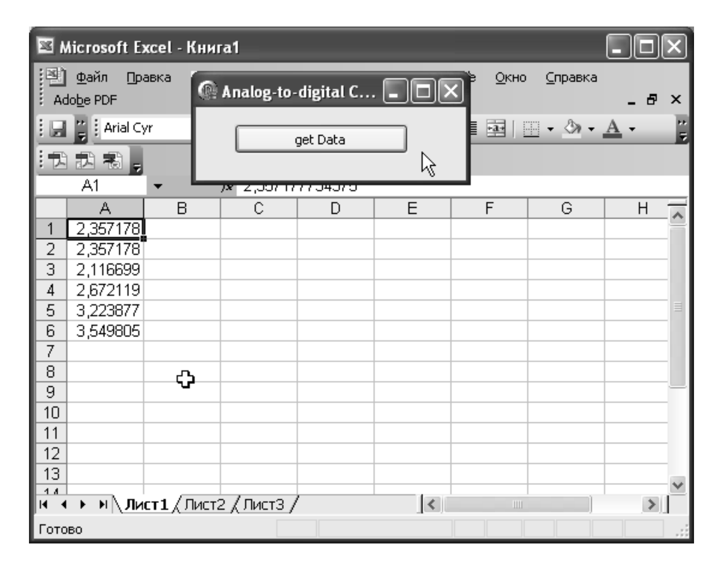


Рис. 3.11
Вид окна работающего приложения

Во многих случаях результаты измерений каких-либо физических величин необходимо передавать на удаленный компьютер для их последующей обработки. Такая необходимость возникает при лабораторных исследованиях, при снятии данных с изолированных систем, в промышленности и т. д.

Для этих целей проще всего использовать так называемую «клиент-серверную» архитектуру. Обычно клиент в этой конфигурации передает данные серверу для дальнейшей обработки или же запрашивает данные с централизованного хранилища данных сервера. Конкретная реализация такой системы определяется в зависимости от требований к измерительной системе.





В нашем следующем примере мы разработаем такую простейшую систему, при этом программа-клиент будет передавать по сети данные аналого-цифрового преобразования на сервер. Несмотря на кажущуюся сложность реализации такой системы, мы разработаем ее максимально простыми средствами, предоставляемыми нам средой разработки Delphi 2007.

Вначале посмотрим в общих чертах, как должна работать наша система (рис. 3.12).

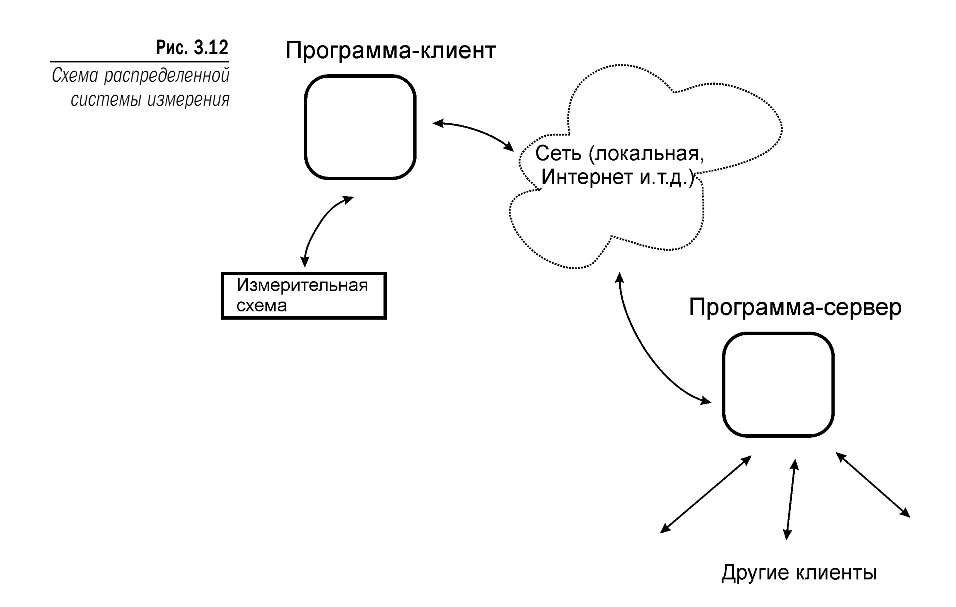


Рис. 3.12
Схема распределенной системы измерения

По этой схеме программа-клиент, получая данные от измерительной системы, отправляет их программе-серверу, работающей на удаленной машине. Клиентский компьютер, на котором работает программа-клиент и удаленный сервер должны взаимодействовать между собой посредством какого-либо сетевого протокола. Наибольшее распространение получил сетевой протокол (или более точно, стек сетевых протоколов), известный под названием TCP/IP. На основе этого протокола построено большинство локальных и глобальных сетей, включая Интернет.

Разработаем распределенную систему передачи данных с аналого-цифрового преобразователя на удаленный хост, на котором запущена программа-сервер. В качестве прототипа клиентской программы будем использовать программу из предыдущего примера, которую модифицируем для работы в сети TCP. Кроме того, разработаем программу-сервер, принимающую и отображающую данные измерений.

Обе программы, как и предыдущие, разработаем в среде Delphi 2007. Начнем с программы-клиента.

Создадим каркас графического приложения и поместим на него компонент **TTcpClient** из закладки Internet. Кроме того, для периодической отправки результатов измерения на TCP-сервер, поместим на форму компонент таймера **TTimer** (закладка System). Собственно, больше никаких компонентов на форме и не будет. На рис. 3.13 показано окно конструктора.

С помощью программы-обработчика таймера входной сигнал аналого-цифрового преобразователя будет считываться и преобразовываться в цифровой код каждые 10 секунд, а затем отправляться по сети с помощью компонента `TcpClient`.

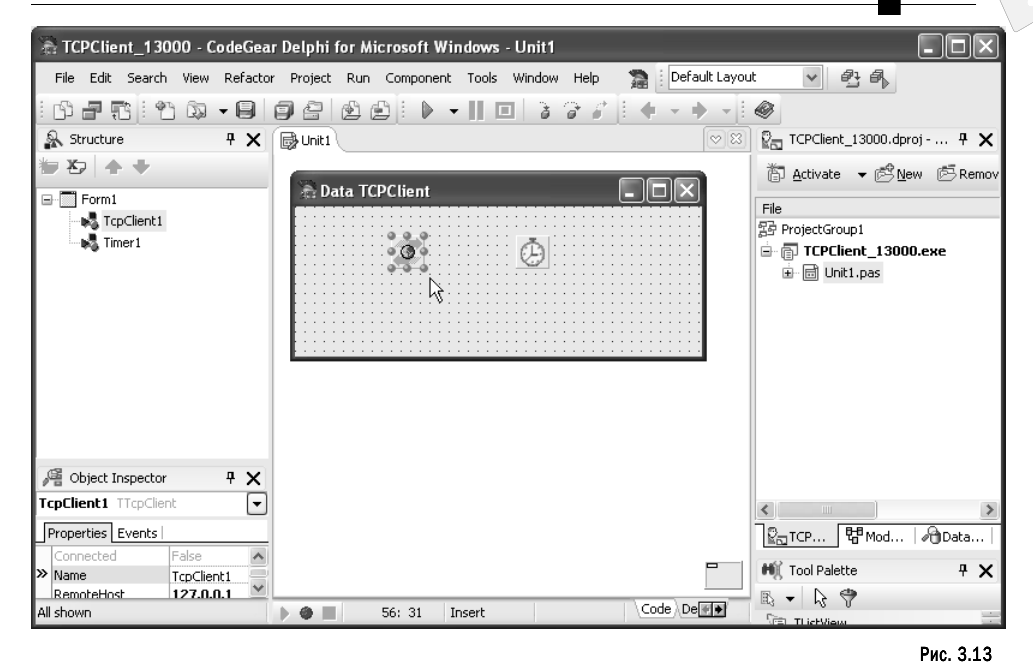


Рис. 3.13

Вид окна конструктора приложения

Алгоритм преобразования данных для удобства мы оформим в виде отдельной процедуры с именем `getData`. Вот ее исходный текст:

```
function TForm1.getData: Real;
const
 DATA = #$378;
 STATUS = #$379;
begin
 asm
  push ebx

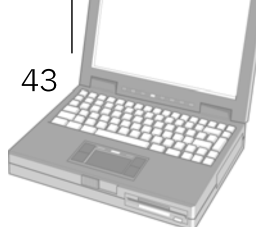
   mov  dx, DATA
   xor  ax, ax
   bts  ax, 7
   out  dx, al

   btr  ax, 7
   out  dx, al

   mov  bx, 15
 @next:

   xor  ax, ax
   mov  dx, DATA
   btr  ax, 6
   out  dx, al


   mov  dx, STATUS
```

```
   in   al, dx
   bt   ax, 3
   rcl  cx, 1

   mov  dx, DATA
   bts  ax, 6
   out  dx, al

   dec  bx
   jnz  @next

   mov  dx, DATA
   bts  ax, 7
   out  dx, al

   pop  ebx
   and  cx, 0FFFh
   mov  word ptr binResult, cx
 end;
 total:= binResult*5.0 / 4096;
 Result:= total;
end;
```

Теперь проанализируем, как данные передаются по сети. Не вдаваясь в подробности, поскольку это тема отдельной книги, рассмотрим вкратце, как происходит взаимодействие программ по сети. Для взаимодействия по сети TCP приложения используют так называемый механизм сокетов (гнезд). Каждый сокет представляет собой пару «интернет–адрес/порт», которая однозначно определяет адресата сетевого сообщения. Интернет-адрес или, по-другому, IP-адрес, представляет собой 32-битовое беззнаковое целое число, которое часто представляют в более удобочитаемом виде в точечном формате, например, 192.163.10.45. Для большинства пользователей Интернет такое даже представление было бы слишком громоздким и неинформативным, поэтому большинству существующих интернет-адресов поставлены в соответствие так называемые имена узлов, например, www.microsoft.com. Обмен данными в сетях происходит с использованием 32-разрядных адресов, поэтому каждое имя сетевого узла должно быть предварительно преобразовано в соответствующий IP-адрес, что выполняется с помощью специально настроенных серверов имен.

В нашем примере мы будем использовать текстовое представление IP-адресов. Для разработки и проверки работы нашей системы можно как приложение-клиент, так и приложение-сервер запускать на одной и той же машине, используя так называемый интерфейс обратной связи, имеющий адрес 127.0.0.1. Кроме IP-адреса, приложения должны знать порт, через который выполняется обмен данными. Порт представляет собой беззнаковое число в диапазоне 0-65535. Пользовательские программы обычно используют номера портов со второго десятка тысяч, хотя могут использоваться и более низкие значения, важно лишь, чтобы программа не занимала уже используемые номера. В нашем случае в качестве значения порта выберем 13000. Приложение-клиент будет отправлять данные серверу по адресу 127.0.0.1 на порт 13000, а сервер будет работать в режиме «прослушивания» (listening) на этом порту и отслеживать запросы на соединения. При поступлении запроса на соединение от клиента сервер создает отдельный канал для обмена данными, продолжая прослушивать порт 13000.

Для того чтобы данные клиента достигли адресата, необходимо правильно сконфигурировать экземпляр `TcpClient1` компонента **TcpClient**. В окне свойств данного компонента необходимо установить как минимум два параметра: Интернет-адрес хоста и порт (рис. 3.14).

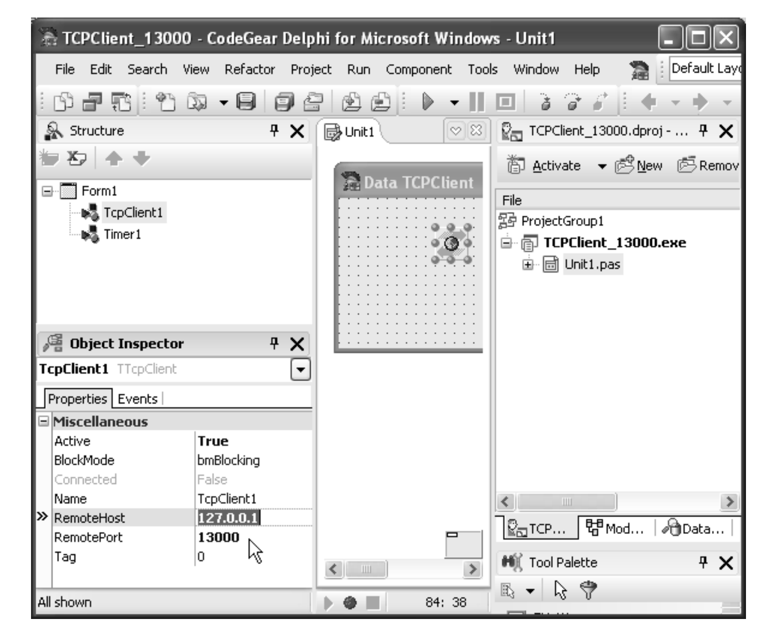


Рис. 3.14
Установка параметров сетевого клиента

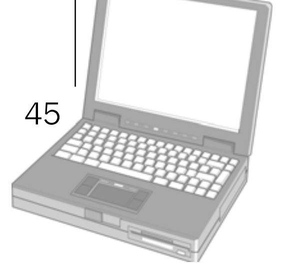
Здесь Интернет-адрес 127.0.0.1 устанавливается в поле **RemoteHost**, а значение порта 13000 – в поле **RemotePort**. В проекте мы будем использовать так называемые блокирующие или, как их еще называют, «ожидающие» сокеты. Это означает, что TCP-сервер может начинать обработку следующего соединения только по завершении обработки текущего (в отличие от неблокирующих или асинхронных сокетов, в которых может обрабатываться одновременно несколько соединений).

При программировании сетевых приложений вручную, с использованием библиотеки Winsock, требуется выполнить множество настроек. В нашем случае, что очень удобно, компонент `TcpClient1` инкапсулирует многие функции, выполняя их автоматически для нас, поэтому единственное, что остается в этом случае, – только отправить данные серверу.

Последнее, что касается настроек компонента `TcpClient1`: при создании окна приложения необходимо активизировать компонент, установив его свойство **Active** в значение `True`, а при закрытии приложения – деактивировать компонент, установив это же свойство в `False`. Кроме того, в силу особенностей работы данного компонента каждый раз перед отправкой данных необходимо свойство **Active** устанавливать в `True`.

Основную работу в программе выполняет функция-обработчик события таймера, исходный текст которой показан далее:

```
procedure TForm1.Timer1Timer(Sender: TObject);
var
 s1: String;
```



```
 fres: Real;
begin
 if TcpClient1.Active = false then
  TcpClient1.Active:= True;
 if TcpClient1.Connected = False then
      TcpClient1.Connect();
 fres:= getData();

 s1:= 'A-D Conversion result:  ' + FloatToStr(fres) + 'V';
 TcpClient1.Sendln (s1);
 if TcpClient1.Connected = True then
   TcpClient1.Disconnect();
end;
```

После проверок состояния сокета данные вычисляются функцией `getData` и помещаются в переменную `fres`, после чего отправляются серверу с помощью оператора `TcpClient1.SendLn(s1)`.

Полный исходный текст клиентского приложения показан ниже:

```
unit Unit1;

interface

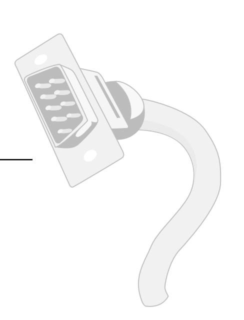
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, Sockets, StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    TcpClient1: TTcpClient;
    Timer1: TTimer;
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure Timer1Timer(Sender: TObject);
    function getData: Real;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  binResult: Word;
  total: Real;

implementation

{$R *.dfm}
```



```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 timer1.Enabled:= false;
 TcpClient1.Active:= False;

end;

procedure TForm1.FormCreate(Sender: TObject);
begin
 timer1.Enabled:= true;
 TcpClient1.Active:= true;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
var
 s1: String;
 fres: Real;
begin
 if TcpClient1.Active = false then
  TcpClient1.Active:= True;
 if TcpClient1.Connected = False then
      TcpClient1.Connect();
 fres:= getData();

 s1:= 'A-D Conversion result:  ' + FloatToStr(fres) + 'V';
 TcpClient1.Sendln (s1);
 if TcpClient1.Connected = True then
   TcpClient1.Disconnect();
end;

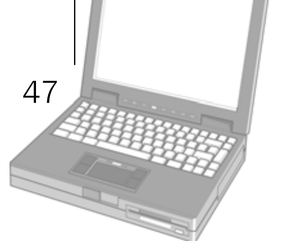
function TForm1.getData: Real;
const
 DATA = #$378;
 STATUS = #$379;
begin
 asm
  push ebx

   mov  dx, DATA
   xor  ax, ax
   bts  ax, 7
   out  dx, al

   btr  ax, 7
   out  dx, al

   mov  bx, 15
 @next:

   xor  ax, ax
```

```
   mov  dx, DATA
   btr  ax, 6
   out  dx, al


   mov  dx, STATUS
   in   al, dx
   bt   ax, 3
   rcl  cx, 1

   mov  dx, DATA
   bts  ax, 6
   out  dx, al

   dec  bx
   jnz  @next

   mov  dx, DATA
   bts  ax, 7
   out  dx, al

   pop  ebx
   and  cx, 0FFFh
   mov  word ptr binResult, cx
 end;
 total:= binResult*5.0 / 4096;
 Result:= total;
end;

end.
```

Создадим теперь программу-сервер, принимающую данные от программы-клиента по сети. На форме приложения поместим экземпляр компонента **TTcpServer** (закладка Internet палитры компонентов) и компонент текстового редактора **TMemo**. В окне этого редактора будут отображаться с интервалом в 10 секунд результаты измерений. На рис. 3.15 можно увидеть, как выглядит окно конструктора приложения.

Для компонента `TcpServer1` необходимо установить параметр **LocalHost** равным 127.0.0.1, а **LocalPort** – 13000. При этом параметр **BlockMode** должен быть установлен в `bmThreadBlocking` (рис. 3.16).

Для серверного компонента входящие подключения должны обрабатываться функцией-обработчиком события **OnAccept**. Исходный текст этой функции представлен далее:

```
procedure TForm1.TcpServer1Accept(Sender: TObject;
  ClientSocket: TCustomIpClient);
var
 s1: String;
begin
 s1:= ClientSocket.Receiveln();
 Memo1.Lines.Add (s1);
 ClientSocket.Close();
end;
```

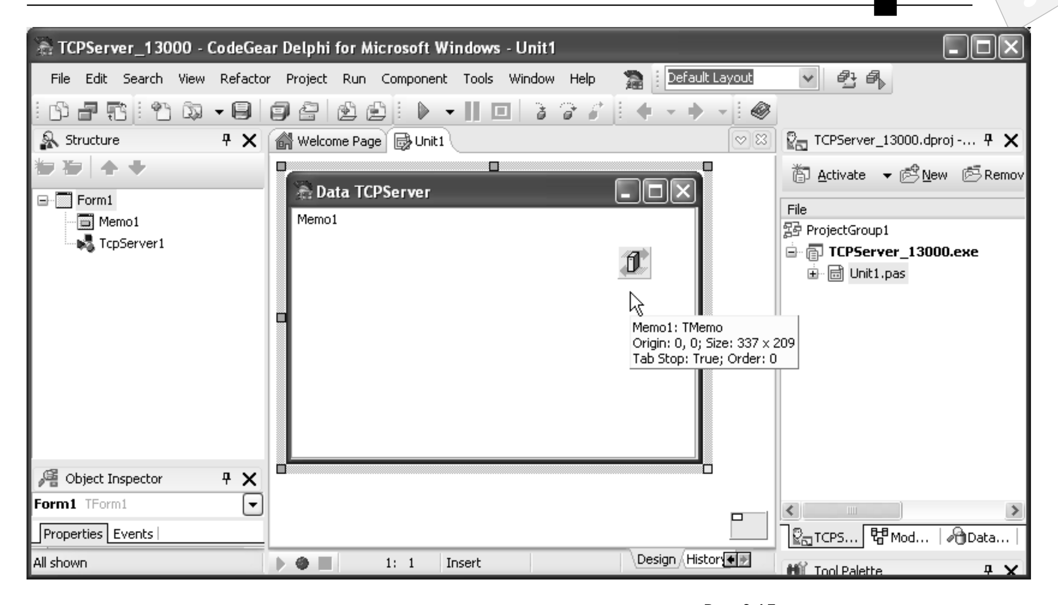


Рис. 3.15

Окно конструктора приложения сервера

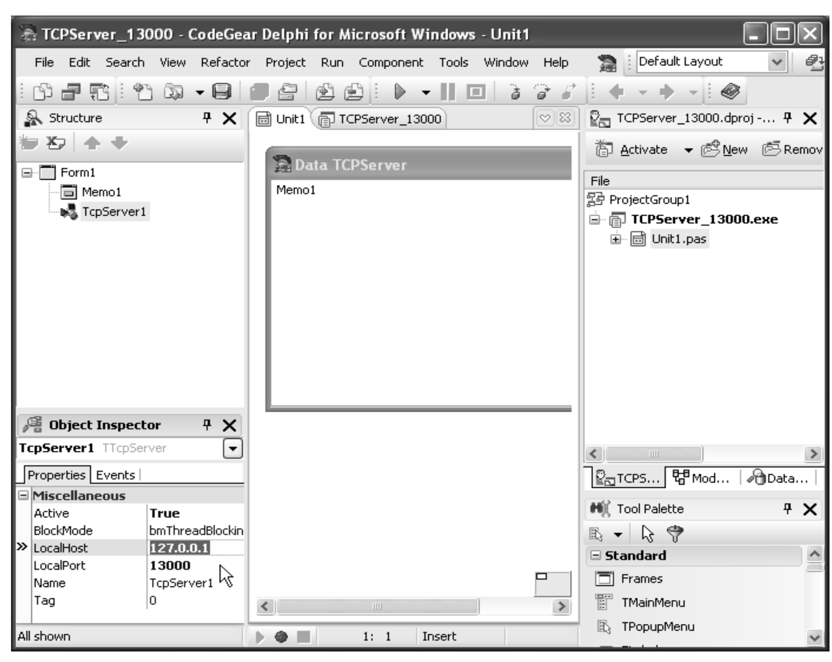
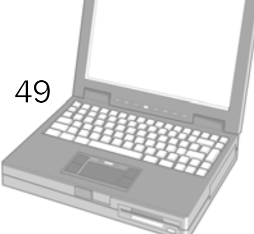


Рис. 3.16

Установка параметров серверного компонента TcpServer1



Полный исходный текст приложения-сервера показан ниже:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, StdCtrls, Sockets;

type
  TForm1 = class(TForm)
    TcpServer1: TTcpServer;
    Memo1: TMemo;
    procedure TcpServer1Accept(Sender: TObject; ClientSocket:
TCustomIpClient);
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  ClientSocket: TCustomIpClient;
implementation

{$R *.dfm}

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  TcpServer1.Active:= False;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
 TcpServer1.Active:= True;
 Memo1.Clear();
end;

procedure TForm1.TcpServer1Accept(Sender: TObject;
  ClientSocket: TCustomIpClient);
var
 s1: String;
begin
 s1:= ClientSocket.Receiveln();
 Memo1.Lines.Add (s1);
```



```
 ClientSocket.Close();
end;

end.
```

Для демонстрации работы нашей системы запустим на компьютере оба приложения – сервер и клиент.

Рисунок 3.17 демонстрирует работающие приложения клиент и сервер.

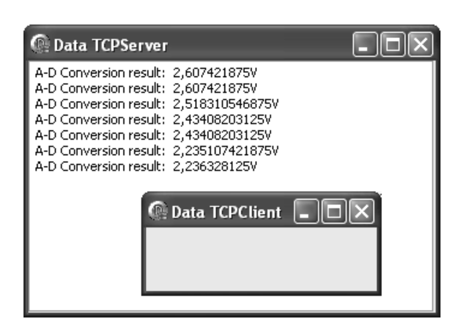


Рис. 3.17
Вид работающих приложений

Заканчивая анализ проекта, напомню, что IP-адреса приложения-клиента и приложения-сервера, как и порты, выбираются в зависимости от конкретной конфигурации вашей сети.

Расширить возможности одноканального преобразователя можно, увеличив количество обрабатываемых входов, например, до 8-ми. Аппаратная часть проекта представлена следующей схемой (рис. 3.18).

Здесь для работы с 8-ю каналами аналоговых данных используется мультиплексор на микросхеме CD4051. Номер канала выбирается установкой одной из 8-ми возможных комбинаций на входах A–C мультиплексора, для чего используются биты 0-2 регистра данных параллельного порта.

Программное обеспечение для 8-канальной системы сбора аналоговых данных создадим с помощью среды программирования Delphi 2007, взяв при этом за основу проект одноканального аналого-цифрового преобразователя с выводом данных в таблицу Excel.

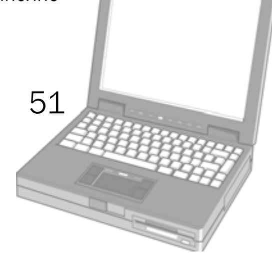
На форму приложения поместим целый ряд компонентов (рис. 3.19).

В левой части формы поместим кнопку выбора канала, с которого будут забираться данные (**Select channel**), счетчик `UpDown1` (компонент **TUpDown**), рядом с которым поместим однострочный редактор `Edit1` (компонент **TEdit**). Наконец, в самом верху поместим метку `Label1` и еще один однострочный редактор `Edit2`, в котором будет отображаться номер выбранного канала.

С компонентом `UpDown1` свяжем редактор `Edit1` (свойство **Associate**) и зададим интервал изменения элемента 0-7 (по числу каналов). При нажатии кнопки **Select channel** номер установленного канала будет зафиксирован в переменной pos, значение которой будет затем использоваться в процедуре преобразования.

В правой части формы поместим кнопку **get Data**, после нажатия которой будет выполнено преобразование по ранее выбранному каналу.

Исходный текст приложения приведен далее:



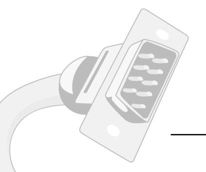


Рис. 3.18

Схема 8-канальной системы сбора аналоговой информации

LPT

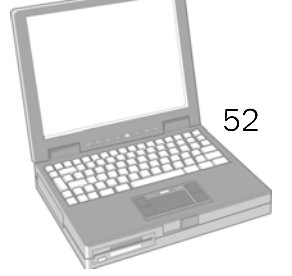
бит 6 рег. 0x378
бит 3 рег. 0x379
бит 7 рег. 0x378
бит 0 рег. 0x378
бит 1 рег. 0x378
бит 2 рег. 0x378

DA2 LTC1286
CLK
D_{OUT}
CS/SHDN
+IN
-IN
V_{REF}
+5V

DA1 4051
X
A
B
C
INH
X0 X1 X2 X3 X4 X5 X6 X7

R1-R8 10K
+5V
VD1-VD8 1N4148

Входные сигналы



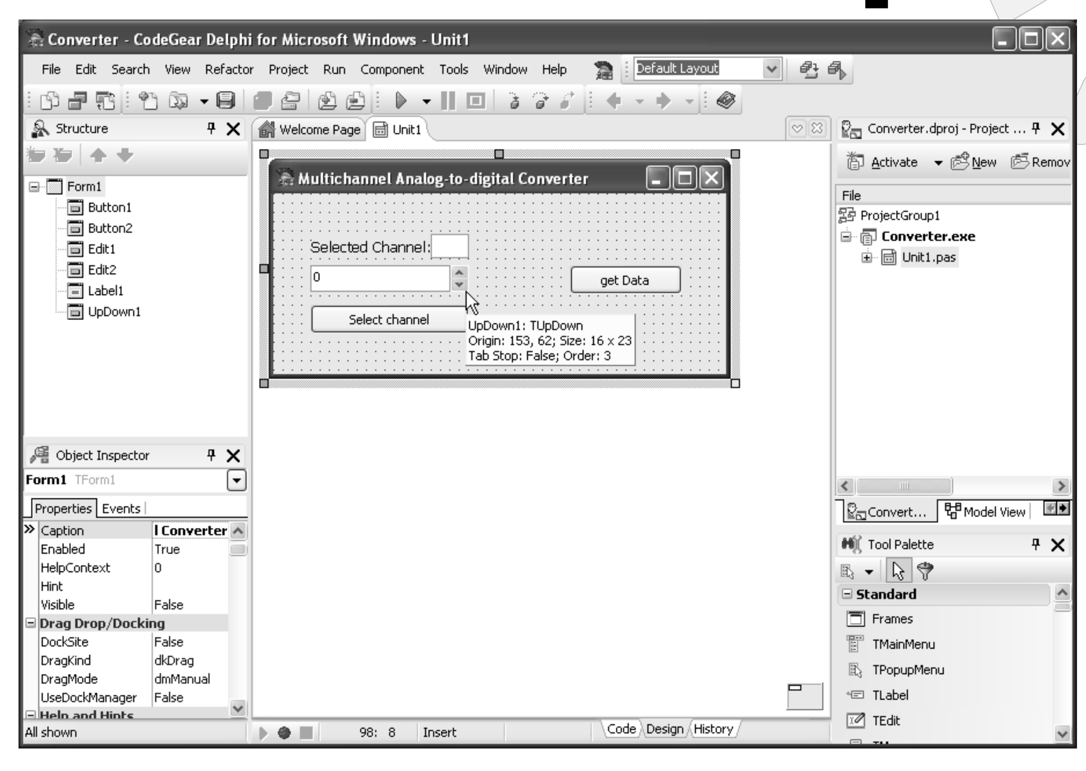


Рис. 3.19

Окно конструктора приложения

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, StdCtrls, ComObj, ComCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    Button2: TButton;
    UpDown1: TUpDown;
    Label1: TLabel;
    Edit2: TEdit;
    procedure Button1Click(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure Button2Click(Sender: TObject);
```

```
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  binResult: Word;
  total: Real;
  XLApp: variant;
  i1: Integer;
  pos: Byte;
implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
const

 DATA = #$378;
 STATUS = #$379;
begin
 asm
   push ebx

   mov  dx, DATA
   xor  ax, ax
   or   al, pos
   bts  ax, 7
   out  dx, al

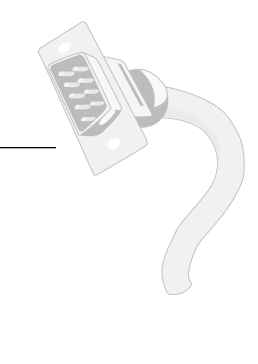
   btr  ax, 7
   out  dx, al

   mov  bx, 15
 @next:

   xor  ax, ax
   or   al, pos
   mov  dx, DATA
   btr  ax, 6
   out  dx, al


   mov  dx, STATUS
   in   al, dx
   bt   ax, 3
   rcl  cx, 1

   mov  dx, DATA
```



```
   bts  ax, 6
   out  dx, al

   dec  bx
   jnz  @next

   mov  dx, DATA
   bts  ax, 7
   out  dx, al

   pop  ebx
   and  cx, 0FFFh
   mov  word ptr binResult, cx

  end;
 total:= binResult*5.0 / 4096.0;

 XLApp.ActiveSheet.cells.item[i1, 1].value:= total;
 Inc(i1);

 end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Edit2.Text:= IntToStr(UpDown1.Position);
  pos:= UpDown1.Position;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
 XLApp := CreateOleObject("Excel.Application");
 XLApp.Workbooks.Add;
 XLApp.Visible := True;
 i1:= 1;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
 if not VarIsEmpty(XLApp) then
  begin
    XLApp.DisplayAlerts := False;
    XLApp.Quit;
  end;
end;
end.
```

Точно так же можно организовать и распределенную многоканальную систему обработки аналоговой информации в формате «клиент-сервер» для работы в сети. В этом случае для обработки данных со всех 8-ми каналов можно использовать таймер, переключающий через определенные интервалы времени каналы.

Рассмотрим одно из многочисленных применений аналого-цифрового преобразования, а именно измерение температуры физических объектов. Если не требуется высокая точность, то в качестве датчика температуры можно использовать полупроводниковый аналоговый трехвыводной датчик типа LM335. Выходным сигналом такого датчика является напряжение, линейно зависящее от температуры измеряемого объекта в градусах Кельвина с коэффициентом 10 мВ/К°. Высокая линейность позволяет обойтись без специальных алгоритмов линеаризации и табличных вычислений, к тому же такой датчик в простейшей конфигурации требует наличия одного внешнего резистора.

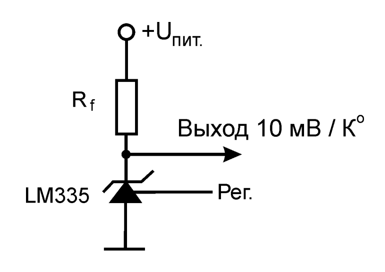


Рис. 3.20
Базовая схема подключения полупроводникового датчика LM335

Базовая схема подключения такого датчика показана на рис. 3.20.

Напряжение питания $U_{пит}$ датчика может изменяться в довольно широком диапазоне, но во многих приложениях удобно использовать +5В. Для работы датчика требуется ток смещения, который выбирают равным порядка единиц миллиампер. В первом приближении величину резистора R_f несложно просчитать, используя значение выходного напряжения датчика при определенной температуре, например, при 273° К (0°С).

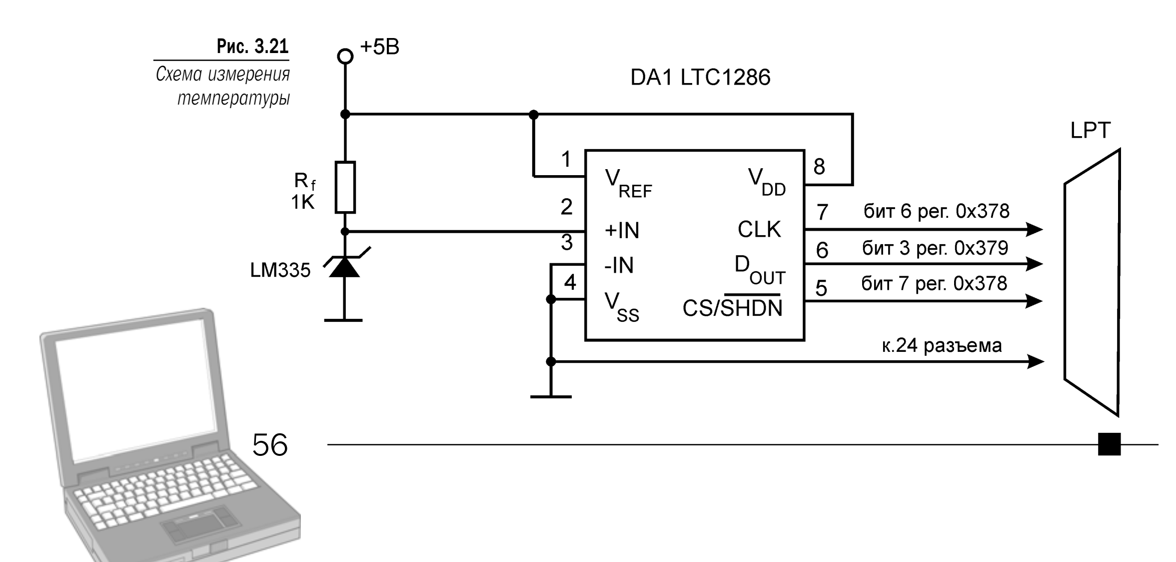
Например, если выбирается ток смещения датчика, равный 1 мА, а напряжение питания равно +5 В, то значение резистора R_f в килоомах будет равно (5 – 2.73)В / 1мА = 2.27 К, т. е. можно взять ближайшее стандартное значение, равное 2.2 К. Здесь мы использовали тот факт, что при 0 °С на выходе датчика LM335 будет напряжение, равное около 10 мА · 273 = 2.73 В.

Базовая схема обладает целым рядом недостатков, которые можно свести к минимуму, применив более сложные схемы подключения датчика, но даже в такой конфигурации можно обеспечить вполне удовлетворительные результаты измерений.

Посмотрим, как можно реализовать простую систему измерения температуры, управляемую с помощью параллельного порта ПК. Аппаратную часть такого проекта реализовать несложно, подсоединив датчик температуры LM335 в базовом включении ко входу аналого-цифрового преобразователя, рассмотренного в начале раздела. На рис. 3.21 показана схема измерения температуры.

При сборке этой схемы следует предусмотреть цепи фильтрации помех и возможную нестабильность источника питания. Лучше всего для питания датчика использовать отдельный источник опорного тока.

Рис. 3.21
Схема измерения температуры



Измеренное напряжение аналого-цифрового преобразователя будет давать значения в вольтах, поэтому для перевода в градусы необходимо откорректировать формулы в соответствующих местах программного кода.



3.4. Расширения портов ввода-вывода

Количество входных или выходных линий параллельного порта ограничено, чего зачастую недостаточно для систем измерения и контроля. Рассмотрим, как решаются такие проблемы. Для расширения количества входных линий параллельного порта можно использовать мультиплексирование, применив цифровые микросхемы, например, 74HC4051. Один из таких вариантов был рассмотрен при проектировании 8-канальной системы обработки аналоговых данных (см. рис. 3.18).

С некоторыми доработками эту схему можно применить для считывания 8-цифровых сигналов (рис. 3.22).

Как видно из схемы, для формирования цифрового сигнала на выходе X мультиплексора установлен триггер Шмита на микросхеме DD1. Поскольку используется инвертор, то и сигнал в 3-м разряде регистра статуса 0x378 также будет инвертирован. Можно обработать этот разряд так, каким он поступает, или включить последовательно с первым инвертором второй.

Для считывания цифрового сигнала с любого из каналов можно разработать несложное приложение в Delphi 2007. Ниже показано окно конструктора приложения (рис. 3.23).

Приложение работает следующим образом: выбрав канал, следует нажать кнопку **Get Data**, в результате чего в окне с меткой `Data` отобразится значение 0 или 1 цифровых данных считываемого канала.

Вот исходный текст приложения:

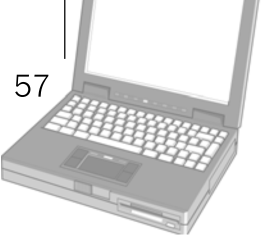
```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, StdCtrls, ComCtrls;

type
  TForm1 = class(TForm)
    UpDown1: TUpDown;
    Edit1: TEdit;
    Button1: TButton;
    Edit2: TEdit;
    Label1: TLabel;
    Label2: TLabel;

    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
```



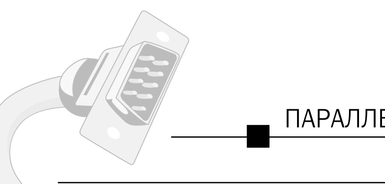


Рис. 3.22

Схема 8-канальной системы сбора цифровой информации

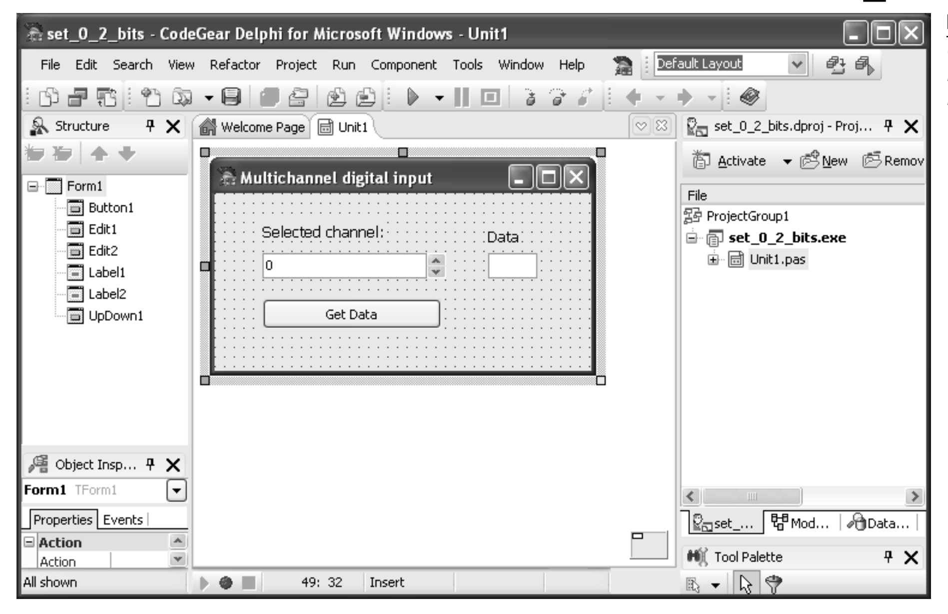


Рис. 3.23
Окно конструктора приложения

```
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
var
 pos, dataBit: Byte;
begin

  pos:= UpDown1.Position;
  asm
    mov   dx, 378h
    xor   al, al
    or    al, pos
    out   dx, al
    //
    inc   dx
    in    al, dx
    shr   al, 3
    mov   dataBit, al
  end;
  Edit2.Text:= IntToStr(dataBit);
end;

end.
```

Рис. 3.24

Схема 16-канальной системы сбора цифровой информации

Количество входных каналов можно увеличить до 16, используя микросхему 74HC4069 (рис. 3.24).

Для обработки данных по 16-ти каналам можно использовать ту же программу, что и для 8-ми каналов, единственное, что нужно сделать – изменить свойство **Max** компонента `UpDown1` с 7 на 15.

3.5. Полезные проекты

В этом разделе мы проанализируем некоторые интересные проекты, которые могут использоваться в радиолюбительской лаборатории. Первый проект, на котором хотелось бы остановиться, – измеритель частоты. Это устройство обычно выполняется автономным и требует немалых усилий по наладке и настройке. Тем не менее, для измерения частоты можно использовать параллельный порт персонального компьютера. Единственный внешний компонент, который может понадобиться, если вы измеряете частоту произвольного сигнала, – формирователь прямоугольного сигнала, который аппаратно может быть реализован на TTL-микросхеме триггера Шмита. Если измеряется частота прямоугольных импульсов, совместимых по уровню с сигналами TTL-логики и имеющих стандартные параметры времени нарастания и спада сигнала, то никаких внешних элементов вообще не требуется.

Следует сказать, что диапазон измерения частот такого компьютерного «частотомера» не очень высок – порядка нескольких десятков килогерц. Это обусловлено в первую очередь тем, что минимальный период времени, который можно отслеживать программными средствами в операционных системах Windows, равен 50 мкс, причем это можно сделать, только написав специальный драйвер, работающий с аппаратными ресурсами системы настолько тесно, насколько это возможно.

При разработке программной части нашего частотомера мы постараемся извлечь максимум из аппаратной части, используя для работы несколько программных потоков.

Вначале остановимся на аппаратной части. Как уже было сказано, если поступающий в параллельный порт сигнал имеет прямоугольную форму, то никаких дополнительных компонентов не требуется. Если же сигнал имеет произвольную форму, то для формирования прямоугольного входного сигнала потребуется простейший триггер Шмита (рис. 3.25).

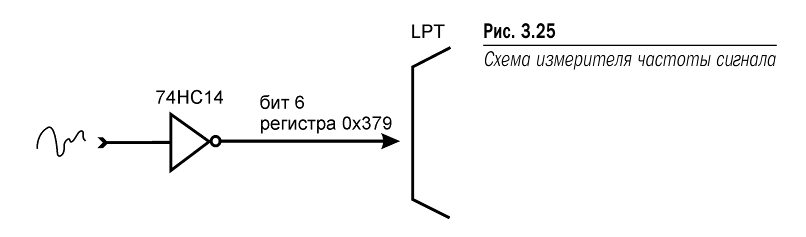


Рис. 3.25
Схема измерителя частоты сигнала

По этой схеме измеряемый входной сигнал приходит на 6-й разряд регистра статуса с адресом 0x379. Для измерения частоты воспользуемся одним из распространенных методов, когда фиксируется стандартный интервал времени, например, 1 секунда, и подсчитывается число перепадов 1-0 (или 0-1) входного сигнала, прошедших за это время (рис. 3.26).

Для большинства радиолюбительских измерений этот метод дает удовлетворительную точность, и мы будем его использовать в нашей разработке. Программную часть проекта, как обычно, разработаем в среде Delphi 2007. Ниже показано окно конструктора (рис. 3.27).



Рис. 3.26
Принцип измерения частоты сигнала

Эталонный интервал времени
1 с
Сигнал

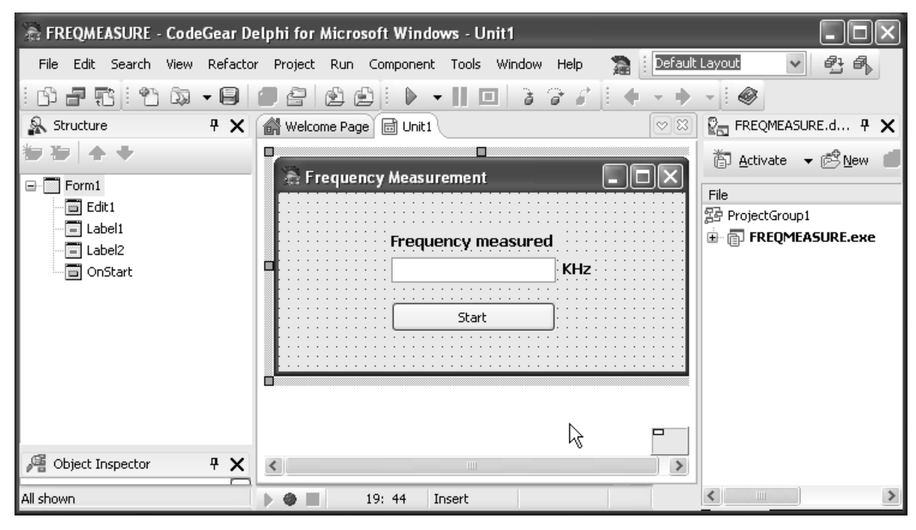


Рис. 3.27
Окно конструктора формы приложения

Как видно из рисунка, на форме размещается копка **OnStart**, окно однострочного редактора текста `Edit1` и две метки – `Label1` и `Label2`. При нажатии на кнопку начинается отсчет интервала времени, равного 1 секунде. Одновременно подсчитывается количество перепадов сигнала 1-0 в 6-м разряде регистра состояния. По истечении интервала в 1 секунду подсчитанное количество перепадов как раз и будет характеризовать частоту сигнала.

Приложения, связанные с обработкой внешних сигналов реального времени, требуют довольно точных интервалов времени, поэтому использовать здесь компонент таймера **TTimer** для формирования временного интервала в 1 секунду будет неэффективным.

Помимо замедления работы приложения, формируемые временные интервалы будут иметь значительную погрешность. В этом случае для формирования измерительного интервала мы будем использовать отдельный поток, который будет создаваться при запуске цикла измерения, и, после односекундного интервала будет давать сигнал об истечении этого интервала в основную программу.

Вот исходный текст приложения:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
```

```
  Dialogs, StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    OnStart: TButton;
    Edit1: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    procedure OnStartClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
type
 TThread1 = class(TThread)
  public
   procedure Execute; override;
 end;

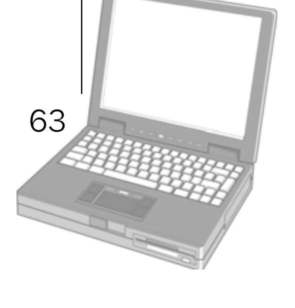
var
  Form1: TForm1;
  Thread1: TThread1;
  Done: Byte;
  cnt: Integer;
  freq: Real;

implementation

{$R *.dfm}

procedure TThread1.Execute;
begin
 Sleep(1000);
 Done:= 1;
end;

procedure TForm1.OnStartClick(Sender: TObject);
begin
 cnt:= 0;
 Done:= 0;
 Thread1:= TThread1.Create(False);
 asm
   clc
   mov  dx, 379h
 @wait_1:
   in   al, dx
   bt   ax, 6
   jnc  @wait_1
   //
```

```
 @wait_0:
   in   al, dx
   bt   ax, 6
   jc   @wait_0
   inc  dword ptr cnt
   cmp  byte ptr Done, 1
   jne  @wait_1
 end;
 Thread1.Terminate();
 freq:= cnt / 1000;
 Edit1.Text:= FloatToStr(freq);
end;
end.
```

В этой программе в обработчике нажатия кнопки создается новый поток `Thread1`, основной функцией которого является ожидание в течение 1 секунды, что реализовано в функции `Execute` потока с помощью функции `Sleep`. По окончании этого интервала функция устанавливает в 1 переменную Done, состояние которой проверяется в блоке `asm ... end`. При установке этой переменной в 1 измерительный цикл заканчивается, и переменная `cnt` будет содержать количество периодов входного сигнала, подсчитанных за 1 с, а это и является частотой сигнала. Как и во всех предыдущих проектах, для запуска приложения необходимо поместить в каталог приложения три файла: porttalk.sys, allowio.exe и start.bat. Последний файл необходимо откорректировать соответствующим образом.

Вторым проектом, который мы рассмотрим, является генератор прямоугольных импульсов. Реализовать генератор прямоугольных импульсов не представляет труда, используя для этого таймер. Необходимо лишь учитывать то, что диапазон такого генератора будет довольно ограниченным, поскольку компонент таймера среды Delphi эффективно работает с разрешением от 1 до 33000 миллисекунд, т. е. наивысшая частота генерации импульсов составит 500 Гц (переключение 1-0 и 0-1 требует двух срабатываний таймера).

Аппаратно такой генератор реализовать несложно, для этого достаточно с определенным интервалом переключать биты регистра данных параллельного порта. Если выходы порта должны работать на мощную нагрузку, то следует ее подключать через буферные усилители-формирователи. Программная часть проекта генератора прямоугольных импульсов представлена далее. Ниже показано, как выглядит окно конструктора (рис. 3.28).

Исходный текст приложения представлен далее, и сложностей для анализа не представляет, поэтому анализировать его мы не будем.

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, StdCtrls, ComCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
```

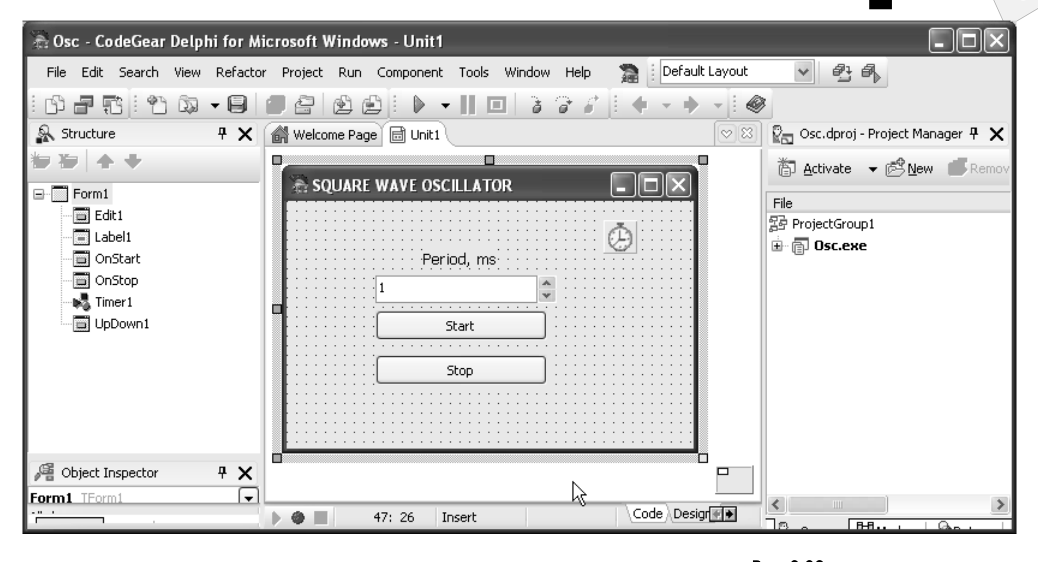


Рис. 3.28
Окно конструктора приложения

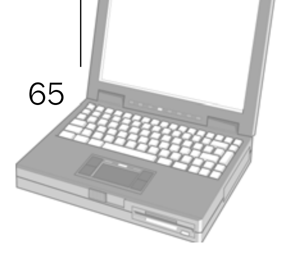
```
    Edit1: TEdit;
    OnStart: TButton;
    Timer1: TTimer;
    UpDown1: TUpDown;
    Label1: TLabel;
    OnStop: TButton;
    procedure Timer1Timer(Sender: TObject);
    procedure OnStartClick(Sender: TObject);
    procedure OnStopClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  period: Integer;

Implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
 timer1.Enabled:= False;
 OnStart.Enabled:= true;
 OnStop.Enabled:= false;
```



```
end;

procedure TForm1.OnStartClick(Sender: TObject);
begin
 OnStart.Enabled:= false;
 OnStop.Enabled:= true;
 period:= UpDown1.Position;
 timer1.Interval:= period;
 timer1.Enabled:= true;
end;

procedure TForm1.OnStopClick(Sender: TObject);
begin
 OnStart.Enabled:= true;
 OnStop.Enabled:= false;
 timer1.Enabled:= false;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
 asm
   mov dx, 378h
   in  al, dx
   not al
   out dx, al
 end;
end;

end.
```

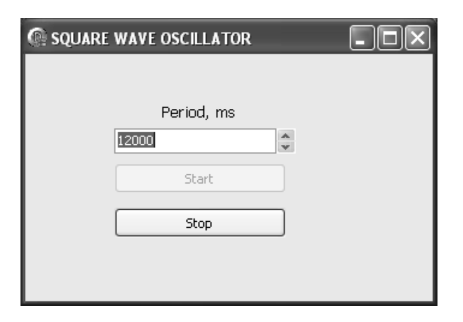


Рис. 3.29 *Вид окна приложения простого генератора импульсов*

Приложение работает следующим образом: в окне редактора следует ввести значение периода сигнала (от 1 до 32000), затем нажать кнопку **Start**. Для остановки генератора импульсов необходимо нажать кнопку **Stop**. Окно работающего приложения продемонстрировано на рис. 3.29.

Проект намного более эффективного и чуть более сложного генератора прямоугольных импульсов, управляемого с параллельного порта, реализуется с использованием нескольких микросхем.

Генератор импульсов собран на популярной микросхеме таймера 555. Особенностью этого генератора является то, что его частотой можно управлять программно, а само устройство подсоединяется к параллельному порту персонального компьютера с работающей операционной системой Windows 2000/XP/2003/Vista.

Для управления генератором в схеме используется цифровой потенциометр MCP41100 фирмы Microchip Technology Inc. с базовым сопротивлением, равным 100 K. Схема устройства показана на рис. 3.30.



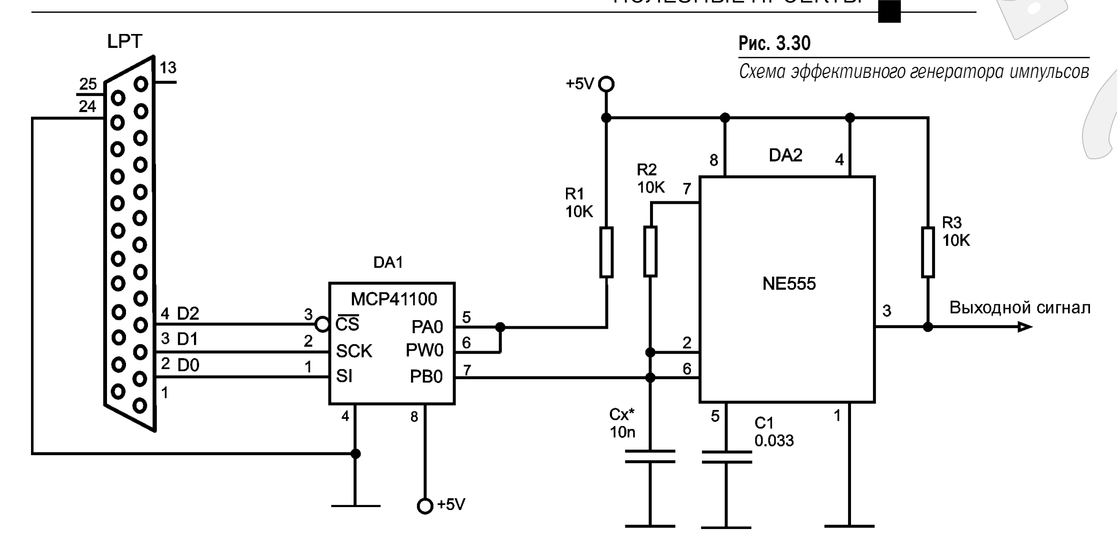


Рис. 3.30

Схема эффективного генератора импульсов

Двоичный код передается из параллельного порта ПК на входы CS, SCK и SI микросхемы MCP41100 с использованием протокола SPI. Команды и данные передаются при установке линии CS в низкий уровень, после чего каждый бит на линии SI запоминается в регистре-защелке цифро-аналогового преобразователя микросхемы MCP41100 по нарастающему фронту сигнала SCK. По завершении передачи данных сигнал CS должен быть установлен в высокий уровень.

Для лучшего понимания работы микросхемы MCP41100 ее функциональную схему можно представить следующим образом (рис. 3.31):

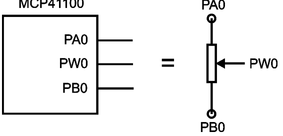


Рис. 3.31

Схема работы цифрового потенциометра

Подключенная в соответствии с этой функциональной схемой микросхема будет работать как потенциометр, причем полярность подключения выводов PA0 и PB0 к внешним цепям не имеет значения. В схеме генератора MCP41100 работает как реостат (рис. 3.32).

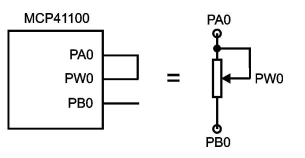


Рис. 3.32

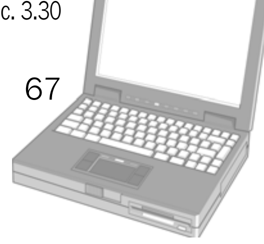
Включение цифрового потенциометра по схеме реостата

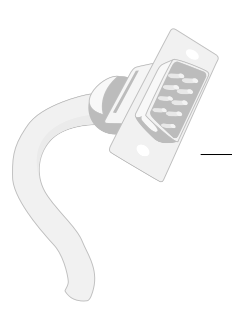
В этом случае вывод PW0 подключен к PA0, как показано на рисунке, хотя может быть соединен и с PB0. Данный цифровой потенциометр является 8-разрядным, что дает 256 возможных градаций сопротивления. Если потенциометр включен по схеме реостата, как в нашем генераторе, то его сопротивление вычисляется по следующей формуле:

$$R_{AB} = D_n \cdot R_{LSB},$$

где R_{AB} — требуемое сопротивление между точками PA0 и PB0, D_n — двоичный код для установки внутреннего цифро-аналогового преобразователя, R_{LSB} — значение (в омах), соответствующее младшему значащему разряду цифрового потенциометра. Поскольку микросхема MCP41100 имеет полное сопротивление, равное 100 K, то величина сопротивления на один бит равна 100000/256 = 390.625 Ом. Например, для того чтобы получить значение сопротивления, равное 10 K, необходимо задать D_n, равное 10000/390.625.

Частота F генератора на 555 таймере применительно к принципиальной схеме на рис. 3.30 рассчитывается по формуле:





$F = 1.44/C_x \cdot (R_1 + R_{AB} + R_2)$.

После окончательного преобразования имеем:

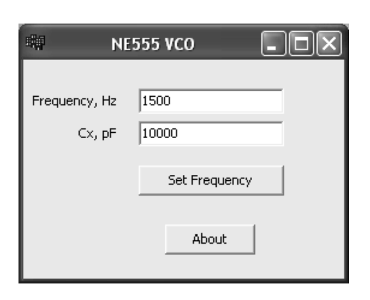
$F = 1.44/C_x \cdot (R_1 + R_2 + D_n \cdot R_{LSB})$.

Таким образом, при фиксированных значениях C_x, R_1, R_2 и R_{AB} частоту генерации можно устанавливать с помощью двоичного кода D_n, записываемого в регистр-защелку цифро-аналогового преобразователя микросхемы MCP41100.

Двоичный код задается следующим образом: вначале микросхеме нужно послать 8-разрядную команду, затем собственно сам 8-разрядный код. Подробно логика работы микросхемы MCP41100 описана в документации фирмы Microchip Corp.

Работоспособность данной схемы была проверена при трех различных значениях емкостей C_x: 1000 pF, 10000 pF и 100000 pF. Программное обеспечение рассчитано для фиксированных значений сопротивлений R_1 и R_2, указанных на схеме, при различных значениях емкости C_x (см. рис. 3.30), при этом частота генерации может изменяться в широком диапазоне частот от десятков герц до десятков килогерц. Хочу подчеркнуть, что программа работает с базовым портом 0x378 параллельного интерфейса (устанавливается в BIOS по умолчанию). Тем не менее, лучше проверить базовый адрес параллельного порта перед запуском программы.

Рис. 3.33
Вид окна работающего приложения

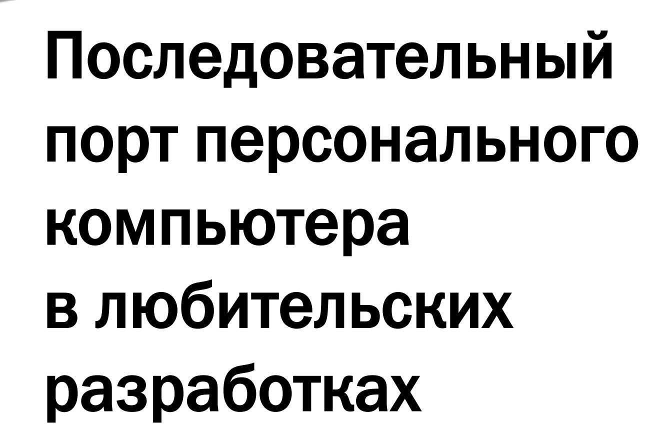


Окно работающего приложения показано на рис. 3.33.

Как видно из рис. 3.33, для установки требуемой частоты необходимо задать емкость конденсатора C_x в пикофарадах и значение частоты в герцах.

Несколько слов о конструкции генератора. В качестве микросхемы таймера 555 можно использовать любое устройство из этой серии из имеющихся на рынке. Для повышения стабильности частоты резисторы R1 и R2 должны иметь минимальные погрешности, а конденсатор Cx желательно выбрать пленочного или металлопленочного типа. Для соединения устройства с параллельным портом ПК желательно использовать плоский кабель длиной не более 30 см. Если же необходим более длинный кабель, то следует применить буферные усилители/формирователи сигналов. Все соединения на плате преобразователя необходимо выполнить проводниками минимальной длины и использовать для питания источник с минимальными пульсациями, либо предусмотреть варианты фильтрации помех по шине питания.

Детальное описание микросхемы MCP41100 можно найти на сайте www.microchip.com фирмы Microchip Technology Inc.



Последовательный порт персонального компьютера в любительских разработках

4 Последовательный порт персонального компьютера в любительских разработках

Последовательный или, как его еще называют, сериальный порт является одним из внешних интерфейсов компьютерных систем, выполняющих обмен данными в соответствии со стандартом RS-232. В большинстве случаев для физической связи устройств, работающих по RS-232, используются 9-контактные соединители, хотя иногда можно встретить и 25-контактные разъемы. Процесс передачи-приема данных по этому стандарту осуществляется последовательно, бит за битом, откуда, собственно, и произошло название «последовательный порт».

Последовательный порт представляет собой один из наиболее ранних интерфейсов обмена данными между устройствами и известен под названием «COM»-порт. Для обмена данными через последовательный порт используются специальные микросхемы, известные под названием UART (Universal Asynchronous Receiver Transmitter, Универсальный Асинхронный Приемо-Передатчик), способные обеспечить скорость обмена данными по последовательному порту до 115 килобит/с (сокращенно kbps), хотя последние разработки позволяют увеличить эту скорость до 460 килобит/с.

В персональном компьютере обычно присутствует не один, а несколько последовательных портов, которым присваиваются следующие аппаратные ресурсы (табл. 4.1).

Таблица 4.1
Аппаратные ресурсы последовательных портов

Наименование порта	Линия прерывания	Базовый адрес регистров
COM1	IRQ4	0x3f8
COM2	IRQ3	0x2f8
COM3	IRQ4	0x3e8
COM4	IRQ3	0x2e8

В настоящее время последовательные порты используются, в основном, для подключения модемов и специальных терминальных устройств, а также для настройки и отладки различного оборудования. Несмотря на то, что более новые скоростные технологии, такие, например, как USB, Bluetooth, Fireware и так далее с каждым днем все больше ограничивают применение последовательных интерфейсов, эта технология все еще широко используется как в уже выпущенном оборудовании, так и в разрабатываемом.

Поговорим немного о принципах обмена данными по последовательному порту. Обмен данных через последовательный порт может осуществляться либо в синхронном, либо в асинхронном режиме.

При синхронном режиме передачи данных передатчик и приемник должны работать на одинаковой частоте. Начало обмена инициируется передатчиком, который посылает приемнику последовательность синхронизирующих бит, за которой следует последовательность бит данных, причем прием-передача данных может осуществляться на значительно более высоких скоростях, чем при асинхронном обмене, поскольку нет необходимости передавать старт-бит, стоп-бит, бит паритета при передаче одного байта. К недостаткам синхронного обмена следует отнести то, что передатчик и приемник должны обладать высокой стабильностью частоты, поскольку даже небольшое рассогласование в частотах приведет к появлению быстро накапливающейся ошибки.

При асинхронной передаче каждому байту данных предшествует старт-бит, за ним следуют биты данных, после которых может передаваться бит паритета (четности) и, в завершение посылки данных, передается стоп-бит, гарантирующий определенную выдержку времени между соседними посылками. Структуру посылки можно представить на рис. 4.1.

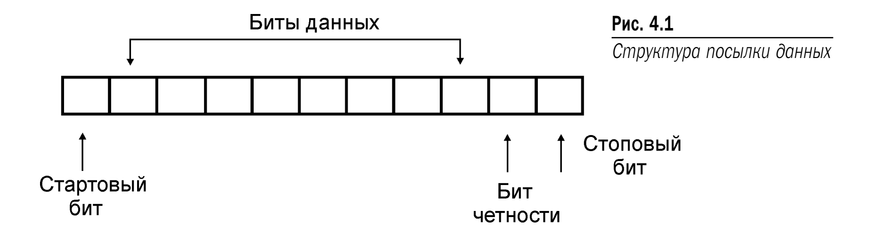


Рис. 4.1
Структура посылки данных

Формат данных последовательного порта обычно представляют в форме записи:

количество_бит_данных–четность–количество_стоповых_бит

Например, запись 8–N–1 интерпретируется как последовательность данных, содержащая 8 бит данных, без контроля четности и с одним стоповым битом.

Запись

8–E–2

интерпретируется как последовательность данных, содержащая 8 бит данных с контролем четности и с двумя стоповыми битами.

Биты данных чаще всего интерпретируются в контексте ASCII-символа. Стартовый бит следующей посылки может передаваться в любой момент времени после окончания стопового бита предыдущей посылки, а количество бит данных может быть равным 5, 6, 7 или 8 бит, в то время как количество стоповых битов может быть 1, 1.5, 2, при этом бит четности может отсутствовать.

Асинхронный обмен данных может осуществляться на одной из скоростей: 50, 75, 110, 150, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600 и 115200 бит/с. Для обмена данными в асинхронном режиме необходимо установить одинаковые параметры приемника и передатчика по скорости, количеству бит данных, проверке четности и количеству стоповых бит.

В компьютерных системах наибольшее распространение получил стандарт асинхронного обмена данными RS-232, который используется как для обмена данными через последовательные COM-порты компьютера, так и для связи автономных периферийных **устройств между собой.**





4.1. Стандарт RS-232

На аппаратном уровне интерфейса RS-232 используются несимметричные приемо-передающие устройства, при этом уровень сигнала отсчитывается относительно общего провода (сигнальной земли). Аппаратная часть интерфейса RS-232 не обеспечивает гальванической развязки устройств по цепям питания (в отличие от другого интерфейса, известного под названием «токовая петля»).

Логические уровни сигналов интерфейса находятся в диапазонах от –3 до –12 В (логическая «1») и от 3 до 12 В (логический «0»). Диапазон –3…+3 В соответствует неактивному состоянию (зона нечувствительности или гистерезис).

В основе асинхронного обмена данными, который используется при функционировании интерфейса RS-232 (как, впрочем, и многих других стандартов), положен принцип обработки отдельных кадров (фреймов) данных, передаваемых между стартовым и стоповым(и) битами.

В интерфейсе RS-232 используется несколько сигнальных линий. Чтобы представить себе процесс передачи-приема данных, посмотрим, что же происходит на любой из линий TD или RD, для чего проанализируем передачу/прием байта данных, который можно представить так, как показано на рис. 4.2.

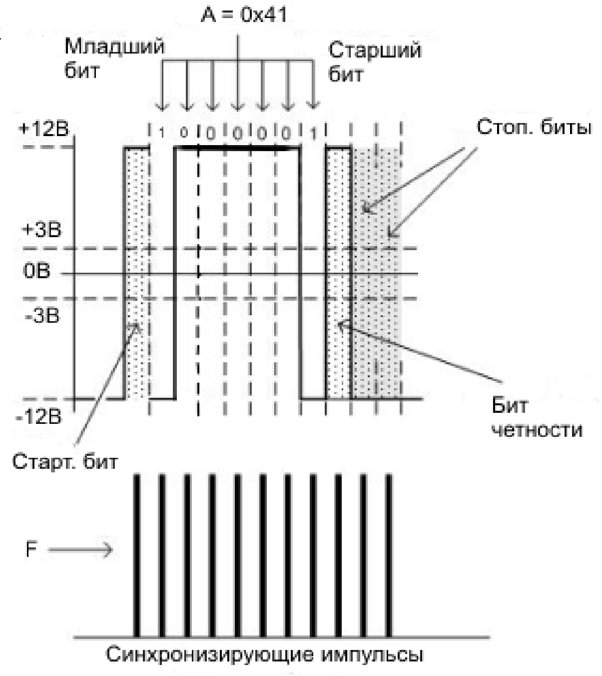


Рис. 4.2
Передача байта данных

На схеме, показанной на рис. 4.2, демонстрируется передача ASCII-символа A в асинхронном режиме: 7 бит данных, бит четности и два стоповых бита. При этом каждому байту данных предшествует стартовый бит, за которым следуют 7 бит данных.

После передачи битов данных передается бит четности (в данном случае, равный 0), и завершают передачу байта данных два стоповых бита, гарантирующих определенную выдержку времени между соседними посылками. Логические уровни сигналов являются инвер-



тированными по отношению к полярности напряжения сигналов, т. е. отрицательным перепадам напряжения соответствует логическая «1», а положительным — логический «0».

Стартовый бит следующей посылки может передаваться в любой момент времени после окончания стопового бита предыдущей посылки, поэтому между двумя последовательными посылками данных могут быть паузы произвольной длительности.

Синхронизация приемника осуществляется стартовым битом передатчика — при этом приемник и передатчик должны работать на одной и той же тактовой частоте F (рис. 4.2). После спада стартового бита передатчика генератор синхроимпульсов приемника сбрасывается и начинает формировать импульсы синхронизации, которые через программируемый делитель частоты и схемы синхронизации управляют приемом данных.

Аппаратно прием и передача данных через интерфейс RS-232 обычно реализуются в многофункциональном контроллере UART. Это общее название для данного типа чипов, хотя выпускается очень много разновидностей этого устройства. Наиболее распространенными контроллерами приема-передачи по последовательному порту являются устройства, совместимые с серией 8250, которая включает чипы 16450, 16550, 16650, 16750.

Для безошибочного приема каждый бит данных должен захватываться в средине интервала времени, когда он является действительным (рис. 4.2). С возрастанием скорости передачи синхронизация передатчика и приемника усложняется — начинают сказываться паразитные емкости электронных узлов, затягивающие фронты синхроимпульсов, а также распределенная индуктивность физической линии передачи.

На практике интерфейс RS-232 реализован как группа сигнальных линий:

- SG (Signal Ground) — сигнальная земля (относительно нее ведется отсчет уровней сигналов);
- TD (Transmit Data) — выход передатчика (TXD);
- RD (Receive Data) — вход приемника (RXD);
- RTS (Request–To–Send) — запрос обмена данных, который выставляется контроллером UART последовательного порта;
- CTS (Clear–To–Send) — сигнализирует о готовности модема или терминального устройства к обмену данными;
- DTR (Data Terminal Ready) — сигнализирует модему или другому устройству, что контроллер последовательного порта готов к установке соединения;
- DSR (Data Set Ready) — сигнализирует контроллеру UART последовательного порта на то, что периферийное устройство готово установить соединение;
- DCD (Data Carrier Detect) — наличие входного сигнала от удаленного модема или устройства. Когда модем или другое устройство обнаруживает появление несущей на другом конце линии, то линия становится активной;
- RI (Ring Indicator) — вход индикатора вызова.

Сигналы RD и TD, определяющие прием/передачу данных, мы уже рассмотрели на примере передачи/приема байта данных. Обратите внимание на то, что линии данных являются последовательными, в то время как сигнальные линии RTS, CTS, DCD, DSR, DTR, RTS и RI фактически являются параллельными.

Напомню, что для соединения устройств, работающих через данный интерфейс используется 9-контактный соединитель, выводы которого соответствуют сигналам, указанным в табл. 4.2.

Эти же сигналы присутствуют и на COM-порту компьютера, к которому подсоединено какое-либо устройство, например, модем.

Таблица 4.2
Сигналы и выводы интерфейса RS-232 для 9-контактного соединителя

Номер вывода на разъеме	Сигнал
3	TD
2	RD
7	RTS
8	CTS
6	DSR
5	SG
1	CD
4	DTR
9	RI

Во многих случаях для проверки и тестирования устройств, присоединенных к последовательному порту, используется так называемое «нуль-модемное» соединение. Такой упрощенный интерфейс используется как для целей диагностики, так и для управления системами промышленной автоматики, реализованных на базе специализированных модулей, например, микроконтроллеров.

Схема нуль-модемного соединения приведена на рис. 4.3.

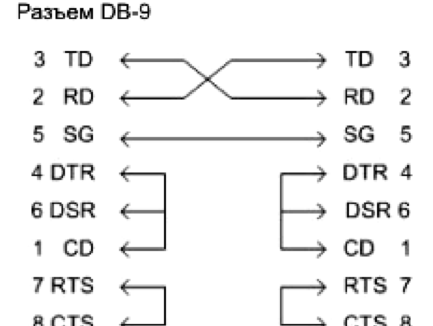


Рис. 4.3
Нуль-модемное соединение

Довольно часто (особенно это касается программистов) возникает необходимость в быстрой проверке функционирования приложений, работающих с последовательными портами. В таких случаях можно воспользоваться так называемым интерфейсом обратной связи (loopback interface), схема которого приведена на рис. 4.4.

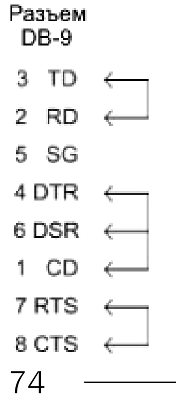


Рис. 4.4
Схема интерфейса обратной связи

Интерфейс RS-232 позволяет управлять потоком данных в зависимости от состояния передатчика и приемника, для чего могут использоваться несколько протоколов обмена данными. Функционирование этих протоколов не касается базовых принципов передачи данных, рассмотренных ранее, их назначение иное — они позволяют синхронизировать устройства, имеющие различные скорости приема/передачи данных, а также различные размеры буферов памяти для хранения данных. Например, принтер, получающий данные через последовательный порт, может откладывать их прием до окончания процесса обработки предыдущего фрагмента данных, сигнализируя об этом передатчику.

Управление потоком данных может осуществляться посредством как аппаратного, так и программного протоколов. Вот наиболее часто используемые протоколы управления обменом данными:

- аппаратный протокол RTS/CTS — используется для соединения принтера с компьютером или для соединения компьютеров (нуль-модемный кабель в этом случае использовать нельзя);
- аппаратный протокол DTR/DSR — функционирует аналогично протоколу RTS/CTS, но использует другие сигналы;
- программный протокол XON/XOFF — применяется в двунаправленных каналах обмена при буферизованном обмене данными. В случае полного заполнения буфера приемника передача данных прекращается и ее возобновление возможно только после обработки данных в буфере приемника. Приемник останавливает передачу данных передатчиком, посылая ему XOFF, и возобновляет ее, инициируя команду XON.
- программный протокол ACK — основан на синхронизации приема одного или нескольких байт данных. Приемник отправляет передатчику команду ACK, в ответ на которую передатчик посылает приемнику байт или группу байтов.

Далее мы рассмотрим практические аспекты применения последовательного порта в различных проектах, которые могут представить интерес для радиолюбителей и всех пользователей ПК, интересующихся созданием систем домашней электроники.

4.2. Устройства измерения и контроля с использованием последовательного порта

С «точки зрения» операционной системы Windows последовательный порт компьютерной системы относится к коммуникационным ресурсам — физическим или логическим устройствам, обеспечивающим обмен данными посредством отдельного двунаправленного асинхронного потока данных.

В практическом плане последовательный порт персонального компьютера может использоваться либо для обмена данными между различными устройствами, либо для управления и контроля другими устройствами. Второй аспект менее известен большинству читателей, поскольку применение последовательного порта для управления другими устройствами требует более сложных технических решений, чем в случае стандартного обмена данными.

Начнем с обмена данными между устройствами. Для операционных систем Windows обмен данными по последовательному порту осуществляется посредством стандартных функций интерфейса WINAPI операционной системы для работы с объектами файловой системы. Прежде всего, замечу, что для программирования операций с COM-портом используются хорошо знакомые нам функции Win API `CreateFile`, `CloseHandle`, `ReadFile` и `WriteFile`, используемые при работе с файлами. Коммуникационные устройства в операционной системе обрабатываются теми же функциями, что и обычные файлы, при этом они имеют специальные имена, например, «COM1» присвоено первому последовательному порту, а «LPT1» — параллельному порту. Так, для открытия коммуникационного устройства COM1 можно использовать функцию `CreateFile`, первым параметром которой будет имя устройства, в данном случае, COM1. Полученный после вызова функции дескриптор устройства (handle) можно использовать при последующих операциях с данными в функциях `ReadFile`, `WriteFile` и `CloseHandle`.





Рассмотрим запись и чтение данных из COM-порта в операционных системах Windows на практических примерах.

В первом примере выполняется передача 8-битового беззнакового числа в последовательный порт. Ниже представлен исходный текст консольного приложения, написанного на языке C++ среды Microsoft Visual Studio .NET.

```
#include <windows.h>
#include <stdio.h>

void main(void)
{
 HANDLE hCom;
 char *pcComPort = "COM1";
 DCB dcb;

 DWORD bytesWritten;
 __int8 i1;

hCom = CreateFile(pcComPort,
                 GENERIC_READ | GENERIC_WRITE,
                 0,
                 NULL,
                 OPEN_EXISTING,
                 0,
                 NULL);
if (hCom == INVALID_HANDLE_VALUE)
{
 printf("COM1 opening error!\n");
 return;
}

GetCommState(hCom, &dcb);
printf("COM1 baud rate is %d\n", dcb.BaudRate);
       {
while (true)
 {
  printf("\nEnter value to be outputted at COM1 port:");
  scanf("%du", &i1);
  WriteFile(hCom,
            &i1,
            sizeof(i1),
            &bytesWritten,
            NULL);
 }
}
CloseHandle(hCom);
return;
}
```



Для читателей, немного знакомых с программированием на C++ в среде Windows, показанный исходный текст не представляет особой сложности. В этой программе вначале открывается файл последовательного порта COM1 с помощью функции WINAPI функции `CreateFile`. При открытии коммуникационных портов в Windows необходимо указывать режим монопольного доступа, установив третий параметр функции `CreateFile` равным 0 и указав атрибут `OPEN_EXISTING`.

В случае успешного открытия порта функция `CreateFile` возвращает дескриптор `hCom` коммуникационного порта, куда и будет записываться введенный с клавиатуры консоли символ. Функция `GetCommState` позволяет получить параметры последовательного порта, такие как скорость обмена, наличие бита четности и т. д. Полученные данные записываются в структуру `dcb` типа `DCB`. В данном случае для контроля скорости обмена данными последовательного порта персонального компьютера на экран консоли выводится значение переменной `dcb.BaudRate`. Думаю, излишне напоминать, что параметры последовательных портов устройств, обменивающихся данными, должны быть одинаковыми, иначе обмен данными будет невозможен.

Далее, программа в цикле `while` выводит в последовательный порт ПК значение, введенное с помощью библиотечной функции `scanf`. Запись байта данных в последовательный порт осуществляется WINAPI-функцией `WriteFile`.

Чтение данных с последовательного порта обычно выполняется в цикле, который может быть запущен как из основного потока программы, так и из отдельного потока. Далее показан исходный текст консольного приложения Windows, выполняющего прием символов из последовательного порта и отображающего полученный результат на строке дисплея.

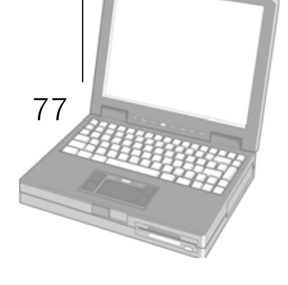
```
#include <windows.h>
#include <stdio.h>

void main(void)
{
 HANDLE hCom;
 char *pcComPort = "COM1";
 DCB dcb;

 DWORD bytesRead;
 char buf[128];

 hCom = CreateFile(pcComPort,
                   GENERIC_READ | GENERIC_WRITE,
                  0,
                  NULL,
                  OPEN_EXISTING,
                  0,
                  NULL);
if (hCom == INVALID_HANDLE_VALUE)
 {
  printf("COM1 opening error!\n");
  return;
 }

GetCommState(hCom, &dcb);
```

```
printf("COM1 baud rate is %d\n", dcb.BaudRate);
printf("Waiting for data from COM1...\n");
do {
    ReadFile(hCom,
             buf,
             sizeof(buf),
             &bytesRead,
             NULL);
    if (bytesRead > 0)
     {
       buf[bytesRead] = '\0';
       printf(buf);
       printf("\n");
     }
 }while (true);
CloseHandle(hCom);
return;
}
```

Здесь, как и в предыдущем Windows-приложении, вначале открывается последовательный порт COM1 (WINAPI-функция `CreateFile`), затем строка из последовательного порта считывается в буфер `buf` функцией `ReadFile`. Если количество `bytesRead` принятых байтов не равно 0, то функция `printf` выводит принятый байт на экран консоли ПК.

При разработке графических приложений в среде Visual Studio. NET для работы с последовательным портом можно использовать также и компонент **SerialPort,** что и будет продемонстрировано в аппаратно-программном проекте простейшей системы контроля температуры с биметаллическим датчиком температуры (термореле).

Аппаратная часть проекта несложная, и ее схема показана на рис. 4.5.

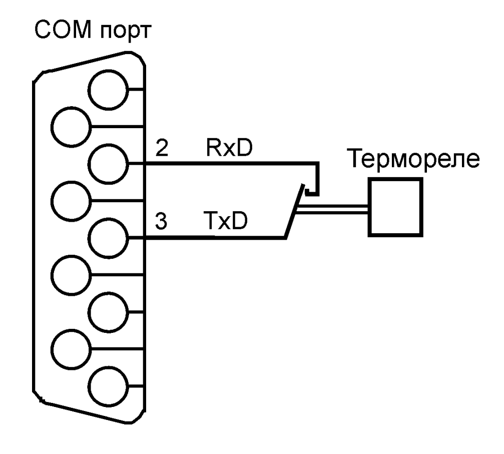


Рис. 4.5
Схема подсоединения датчика температуры к COM-порту

Принцип работы схемы на рис. 4.5 основан на применении замкнутого шлейфа при соединении сигнальных линий TxD и RxD последовательного порта. При температуре ниже срабатывания датчика, в качестве которого используется термореле, TxD и RxD разомкнуты. При достижении определенной, установленной температуры срабатывания контакты реле замы-



каются, соединяя тем самым линии TxD и RxD вместе. Если на компьютере работает какая-либо программа, посылающая, например, звуковой сигнал при замыкании контактов, то это может свидетельствовать о превышении температуры какого-либо физического объекта или, например, воздуха в помещении.

Термореле может располагаться на достаточном удалении от персонального компьютера, но при этом следует учитывать возможность затухания сигнала и завал фронта-спада сигналов из-за наличия распределенного емкостного и омического сопротивлений, которые могут образовывать RC-пары, вызывающие затухания сигнала на частотах выше $1/2\pi RC$. Хотя в такой длинной линии присутствует и определенная индуктивность, но при относительно невысоких частотах передачи сигналов по COM-порту существенного влияния она оказывать не будет. Для надежной работы на удаленном расстоянии лучше всего использовать формирователи сигналов, но это потребует усложнения схемы.

Рассмотрим, как можно реализовать программную часть такого проекта. Для разработки приложения будем использовать среду Microsoft Visual Studio .NET 2005.

Создадим новый проект на языке Visual Basic .NET, выбрав из меню **Visual Basic** тип приложения **Windows Application** (рис. 4.6).

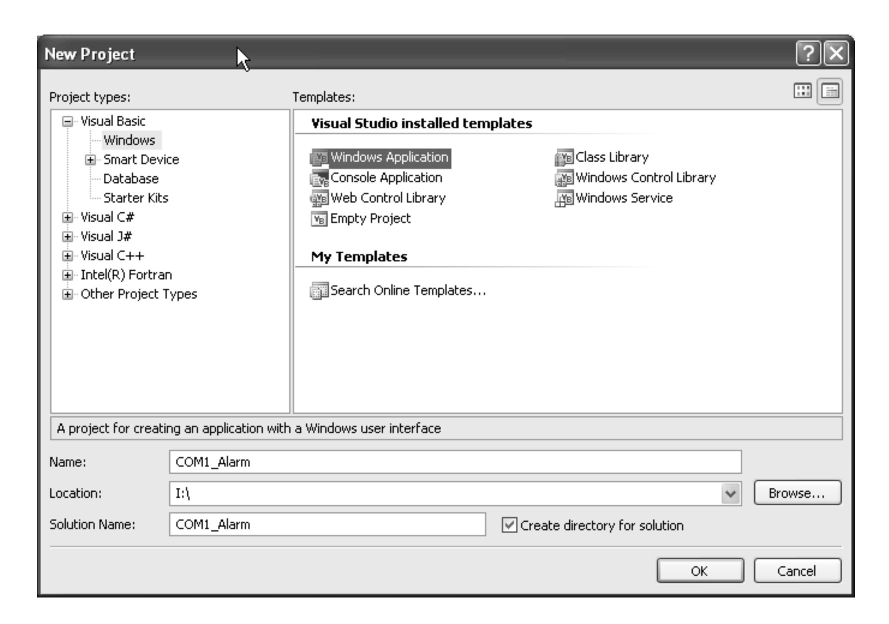


Рис. 4.6

Окно мастера создания нового проекта

Назовем проект COM1_Alarm и после нажатия кнопки **OK** продолжим его настройку. Для этого добавим в проект компоненты **SerialPort** и **Timer** из палитры компонентов **Toolbox** (рис. 4.7).

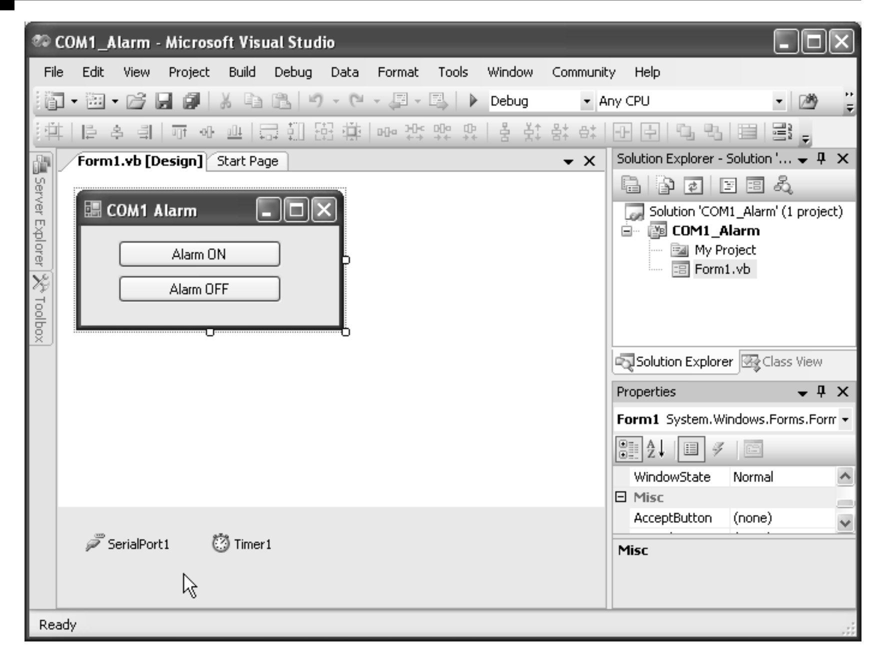
Кроме того, разместим на форме две кнопки с названиями, показанными на рисунке. При нажатии верхней кнопки будет включен таймер, который каждые 5 с будет посылать в последовательный порт сообщение, причем неважно какое, поскольку для нашей программы имеет значение только то, принято сообщение или не принято, т. е. замкнуты контакты термореле или разомкнуты.

При нажатии нижней кнопки работа таймера останавливается, и может в любой момент возобновиться опять верней кнопкой. Посмотрим более детально на назначение и работу программных элементов.



Рис. 4.7

Вид модифицированного окна проекта



Первый компонент, работу которого проанализируем, – **SerialPort.** При перетаскивании компонента в область дизайна был создан экземпляр этого компонента с именем `SerialPort1`. Данный компонент кардинально упрощает работу с COM-портом, поскольку его легко настроить и несложно применить. Установим свойства компонента `SerialPort1` так, как это показано на рис. 4.8.

Большинство настроек система установила самостоятельно (например, скорость обмена в 9600 бод, имя порта (в данном случае, «COM1»), количество бит данных, проверку на четность и т. д.). Эти параметры должны соответствовать установленным для COM-порта в системе Windows, поэтому при необходимости следует изменить параметры для компонента **SerialNet**, установленные средой.

Компонент `SerialPort1` может посылать и принимать данные через последовательный порт. Послать данные несложно, для этого достаточно вызвать один из подходящих методов класса `SerialNet`; что же касается приема данных, то здесь ситуация немного сложнее, поскольку компонент «не знает», когда поступят данные, поэтому для этих целей используется механизм событий. Для компонента **SerialPort** в момент поступления данных в последовательный порт генерируется событие **DataReceived**. Иными словами, мы можем написать функцию-обработчик события **DataReceived** и больше не заботиться о том, как принять данные.

Установим функцию-обработчик (назовем ее `SerialPort1_DataReceived`) события **DataReceived,** как показано на рис. 4.9.

Исходный текст функции-обработчика показан ниже:

```
Private Sub SerialPort1_DataReceived(ByVal sender As System.Object, ByVal
e As System.IO.Ports.SerialDataReceivedEventArgs) Handles
SerialPort1.DataReceived
   Dim readMes As String = SerialPort1.ReadLine()
```

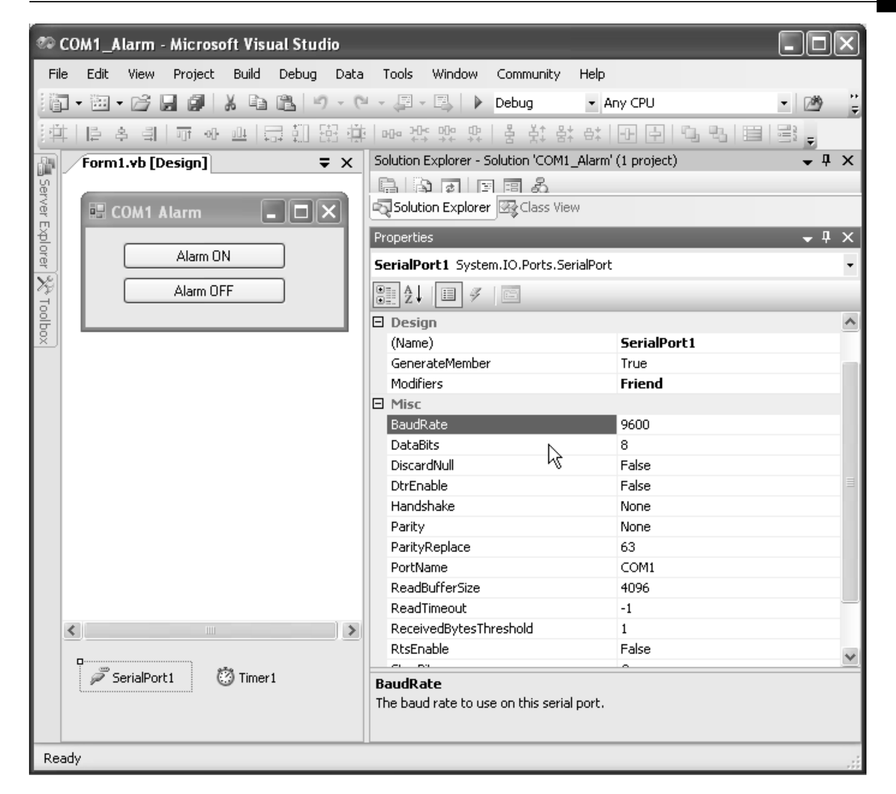


Рис. 4.8

Установка свойств компонента `SerialPort1`

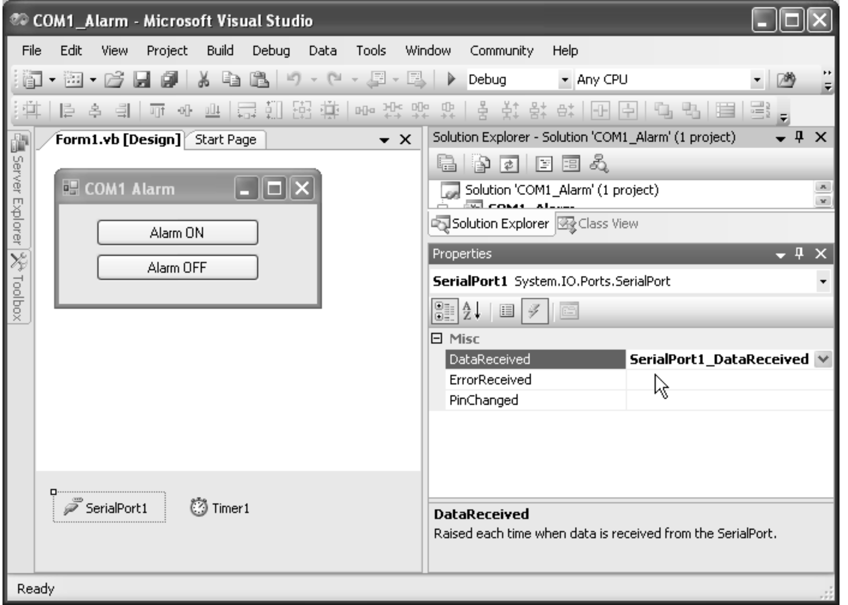


Рис. 4.9

Установка функции-обработчика события `DataReceived`




```
   Beep()
End Sub
```

Функция `SerialPort1_DataReceived` содержит всего два оператора. Первый оператор, `SerialPort1.ReadLine()`, использует метод `ReadLine()` для чтения строки байтов из порта. Здесь этот оператор используется только как демонстрация того, как можно принять данные из COM-порта и дополнительной нагрузки не несет. Второй оператор, `Beep()`, позволяет генерировать звуковой сигнал, когда данные по линии RxD приняты, т. е. датчик сработал. Если линии TxD и RxD разомкнуты, то событие **DataReceived** не генерируется, и звукового сигнала нет.

Посмотрим, как отправить сообщение на COM-порт. Для этого будем использовать экземпляр компонента **Timer** `Timer1`. Во-первых, установим свойства этого компонента (рис. 4.10).

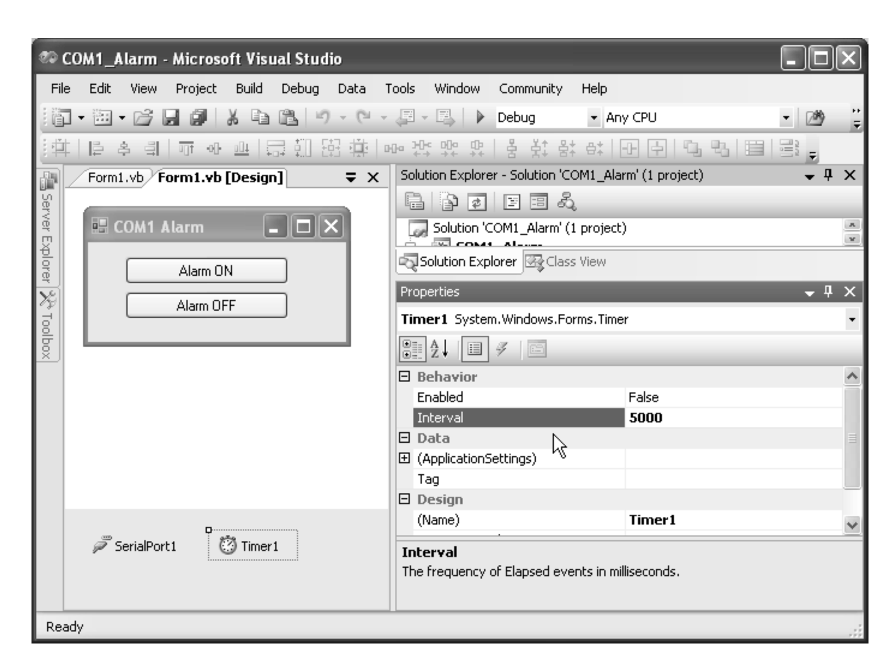


Рис. 4.10
Установка свойств экземпляра компонента `Timer1`

В исходном состоянии (т. е. при запуске приложения) `Timer1` должен быть отключен, для чего его свойство **Enabled** следует установить в **False**. Выберем интервал срабатывания таймера равным 5000 миллисекунд (5 с). Установим функцию-обработчик `Timer1_Tick` события **Tick**, которое в нашем случае будет происходить каждые 5 с (рис. 4.11).

Ниже представлен исходный текст функции-обработчика `Timer1_Tick`:

```
Private Sub Timer1_Tick(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Timer1.Tick
        SerialPort1.WriteLine(Cnt.ToString())
        Cnt = Cnt + 1
End Sub
```

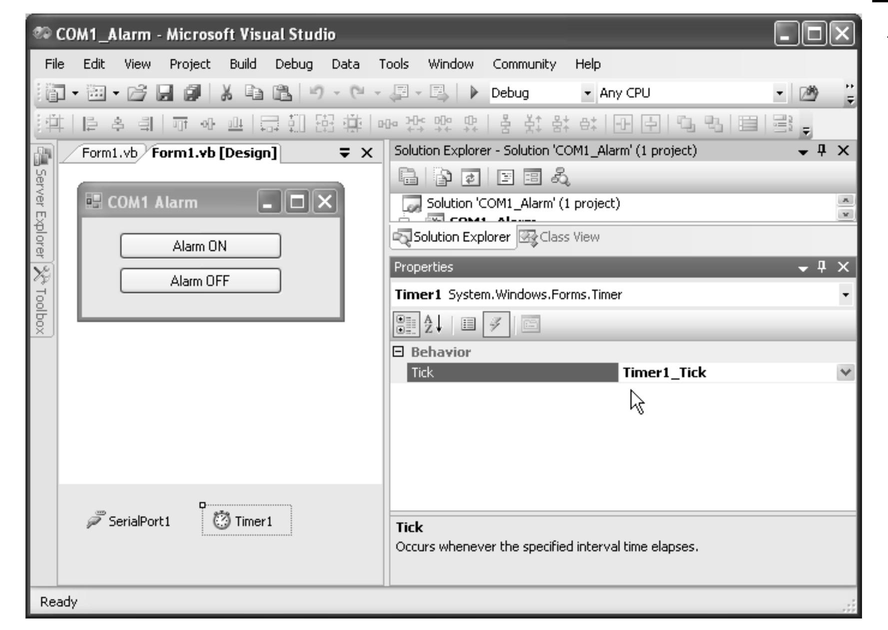


Рис. 4.11
*Установка обработчика события **Tick***

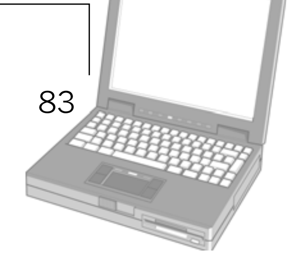
Во время каждого срабатывания таймера в COM-порт посылается значение счетчика, которое затем инкрементируется.

Перед тем как что-либо делать с последовательным портом, его нужно открыть. В терминах операционной системы Windows это означает, что нужно ассоциировать устройство или файл (что, в принципе, одно и то же) с определенными атрибутами и ресурсами, выделенными системой, посредством дескриптора. Для нас важно только то, что перед записью-чтением последовательного порта его нужно открыть. Вполне возможна ситуация, когда порт уже открыт (это возможно, если другая программа уже открыла порт, или, что хуже, произошел какой-то сбой). Попытка открытия порта выполняется при нажатии кнопки **Alarm ON** и реализована следующим фрагментом программного кода:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
        If SerialPort1.IsOpen = False Then
            SerialPort1.Open()
        End If
        Cnt = 0
        Timer1.Enabled = True
        Button2.Enabled = True
        Button1.Enabled = False
    End Sub
```

Мы проанализировали работу отдельных частей программного кода приложения. Далее рассмотрим полный исходный текст приложения:

```
Public Class Form1
    Dim Cnt As Integer
```



```
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
        If SerialPort1.IsOpen = False Then
            SerialPort1.Open()
        End If
        Cnt = 0
        Timer1.Enabled = True
        Button2.Enabled = True
        Button1.Enabled = False
    End Sub

    Private Sub SerialPort1_DataReceived(ByVal sender As System.Object,
ByVal e As System.IO.Ports.SerialDataReceivedEventArgs) Handles
SerialPort1.DataReceived
        Dim readMes As String = SerialPort1.ReadLine()
        Beep()
    End Sub

    Private Sub Form1_Activated(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Activated
        Button1.Enabled = True
        Button2.Enabled = False
    End Sub

    Private Sub Form1_FormClosing(ByVal sender As Object, ByVal e As
System.Windows.Forms.FormClosingEventArgs) Handles Me.FormClosing
        SerialPort1.Close()
        Timer1.Enabled = False
    End Sub

    Private Sub Timer1_Tick(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Timer1.Tick
        SerialPort1.WriteLine(Cnt.ToString())
        Cnt = Cnt + 1
    End Sub

    Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button2.Click
        Timer1.Enabled = False
        Button2.Enabled = False
        Button1.Enabled = True
    End Sub
End Class
```

После успешной компиляции приложения можно приступить к его проверке. Присоедините к COM-порту термореле, как показано на рис. 4.5 (вполне возможно, что ваше устройство будет иметь иную конструкцию, но в любом случае при срабатывании оно должно замыкать линии TxD и RxD), и запустите программу. При нажатии кнопки **Alarm ON** сигнализация включается, и при срабатывании датчика температуры каждые пять секунд будет генерироваться звуковой сигнал.

Следующий пример, который мы рассмотрим, более сложен. Мы разработаем простую систему контроля доступа в помещение на базе инфракрасного пироэлектрического (PIR) датчика перемещения. Как и в предыдущей разработке, мы будем использовать замыкание/размыкание шлейфа, соединяющего сигнальные линии TxD и RxD последовательного порта. При этом замыкание шлейфа будет осуществляться сигналом датчика (уровнем логической 1 на его выходе), что приведет в действие остальную часть схемы. Программная часть проекта довольно сложна, но реализовать ее может программист средней руки, немного разобравшись в работе программного кода. Сама программа реализована как простой веб-сервер, доступ к которому может осуществляться из другой сети (локальной или Интернет), хотя можно настроить сервер и на работу с локальной машиной, где выполняется программа. Рассмотрим наш проект более подробно и начнем с аппаратной части. Схема контроля доступа показана на рис. 4.12.

В схеме могут быть применены пироэлектрические детекторы различных типов, в зависимости от размера помещения и способа размещения самого датчика, а также диаграммы его направленности и максимального расстояния, на котором он способен работать. Большинство PIR-детекторов выпускаются с интегрированными формирователями сигналов, как правило, соответствующими уровням сигналов цифровой TTL-логики. В нашей схеме мы будем использовать PIR-детектор, выходной сигнал которого ($U_{вых}$) становится равным уровню лог.1 при фиксации перемещения объекта. Схема на четырех логических элементах представляет собой формирователь сигнала (элементы DD1.1 и DD1.2) и RS-триггер, который удаляет «дребезг» и открывает/закрывает ключ на биполярном транзисторе Q1. Коллектор транзистора подключен к линии TxD, а эмиттер – к линии RxD последовательного порта. При появлении сигнала высокого логического уровня на выходе PIR-детектора транзистор Q1 открывается, замыкая цепь TxD – RxD. Соответствующее программное обеспечение может фиксировать состояние шлейфа, выдавая соответствующее сообщение.

Программная часть проекта построена как веб-сервер, доступ к которому осуществляется через браузер, например, Internet Explorer, по соответствующему адресу. В этом программном проекте веб-сервер работает на той же машине, к которой подключена схема на рис. 4.12. Веб-сервер представляет собой простейший TCP-сервер, работающий с локальным сетевым адресом компьютера (127.0.0.1), принимая подключения клиентов на порту 8080, хотя можно выбрать и другой порт из диапазона допустимых (стандартный порт с номером 80 лучше не выбирать, если на вашей машине уже установлен какой-нибудь веб-сервер, например, Microsoft IIS). Исходный текст веб-сервера с целью максимального упрощения разработан на Visual Basic .NET, а сам веб-сервер работает в операционных системах Windows. Для разработки веб-сервера можно использовать бесплатные версии среды разработки NET (Express Editions 2005 или 2008).

Разработка веб-сервера в среде Visual Studio .NET 2005 состоит из нескольких шагов.

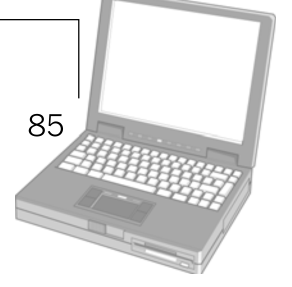
На первом шаге необходимо создать новый проект и выбрать его тип (рис. 4.13).

Как видно из рисунка, выбираем создание консольного приложения в Visual Basic .NET.

После создания проекта приложения в него будет включен шаблон модуля с расширением .vb (в данном случае, это Module1.vb), в который необходимо включить исходный текст нашего сервера (рис. 4.14).

Исходный текст нашего веб-сервера представлен далее:

```
Imports System
Imports System.IO.Ports
```



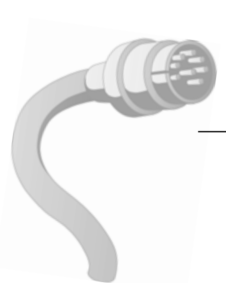
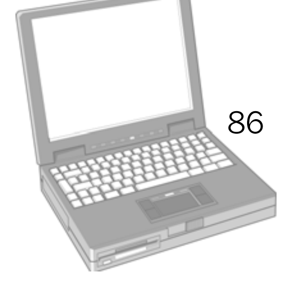


Рис. 4.12

Схема контроля доступа в помещение на пироэлектрическом датчике перемещения

TxD
RxD
Q1
2N3904
R1
10K
DD1.3
DD1.4
DD1.2
DD1.1
74HC132
PIR-детектор
Uвых
+5В



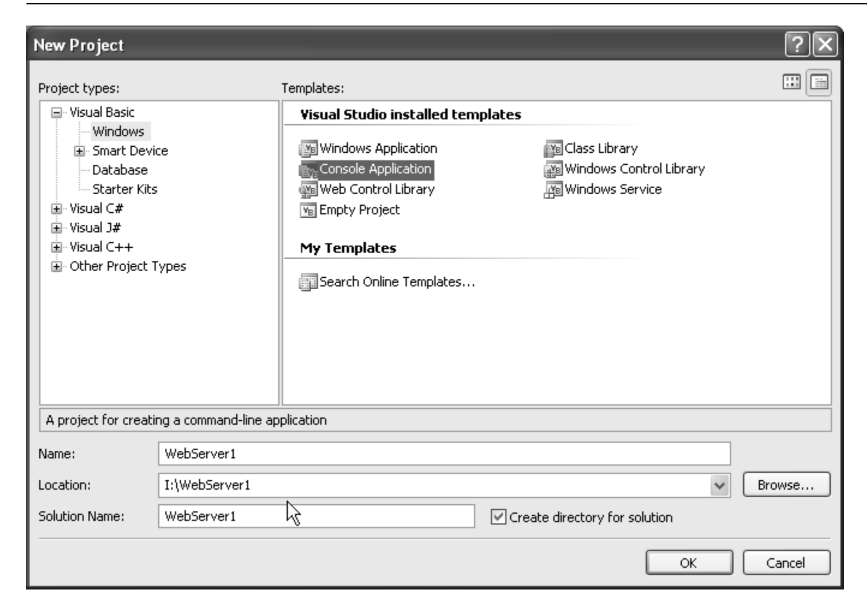


Рис. 4.13

Выбор типа проекта в Visual Studio .NET

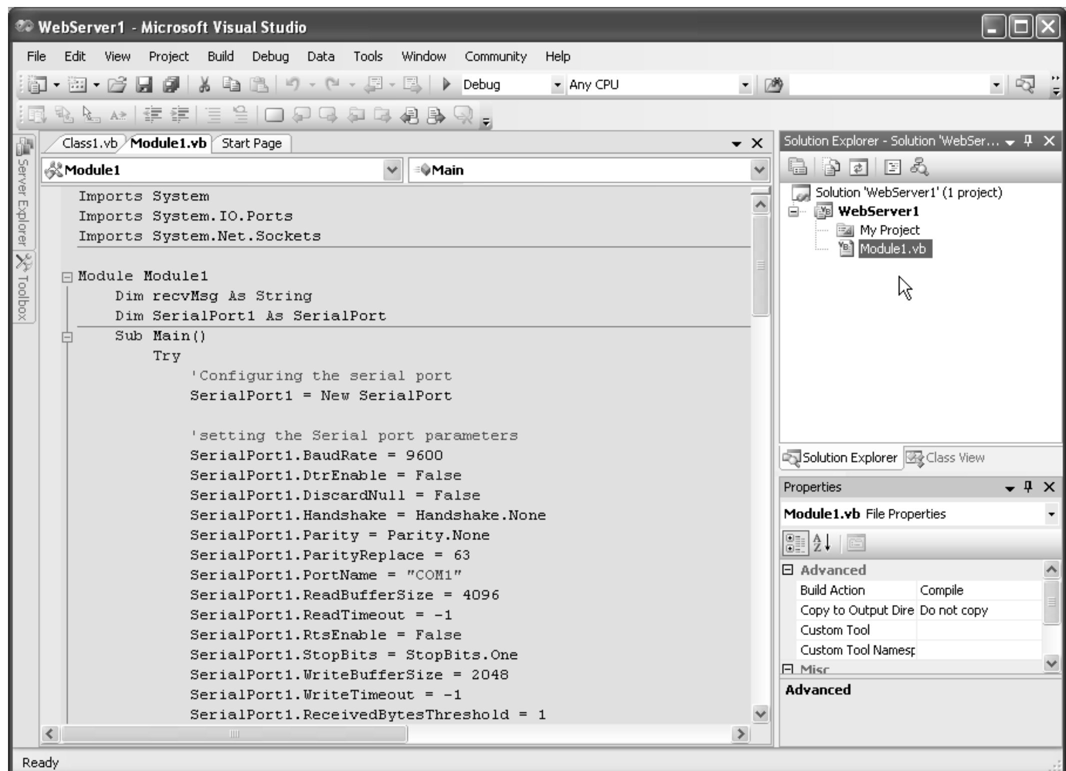
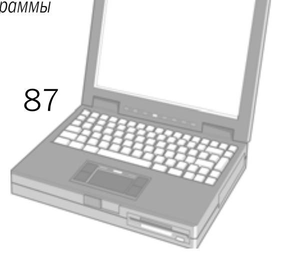


Рис. 4.14

Рабочее окно проекта с исходным текстом программы



```
Imports System.Net.Sockets

Module Module1
 Dim recvMsg As String
 Dim SerialPort1 As SerialPort
 Sub Main()
  Try
   'Configuring the serial port
   SerialPort1 = New SerialPort

   'setting the Serial port parameters
    SerialPort1.BaudRate = 9600
    SerialPort1.DtrEnable = False
    SerialPort1.DiscardNull = False
    SerialPort1.Handshake = Handshake.None
    SerialPort1.Parity = Parity.None
    SerialPort1.ParityReplace = 63
    SerialPort1.PortName = "COM1"
    SerialPort1.ReadBufferSize = 4096
    SerialPort1.ReadTimeout = -1
    SerialPort1.RtsEnable = False
    SerialPort1.StopBits = StopBits.One
    SerialPort1.WriteBufferSize = 2048
    SerialPort1.WriteTimeout = -1
    SerialPort1.ReceivedBytesThreshold = 1

   'Adding the Serial port Event Handler
   AddHandler SerialPort1.DataReceived, New
IO.Ports.SerialDataReceivedEventHandler(AddressOf
SerialPort1_DataReceived)

   'Opening the serial port
   If SerialPort1.IsOpen = False Then
      SerialPort1.Open()
   End If

   ' Configuring the TCP/IP Server
   ' Set the TcpListener on port 8080.
   Dim port As Int32 = 8080
   Dim addr As Net.IPAddress = Net.IPAddress.Parse("127.0.0.1")
   Dim server As New TcpListener(addr, port)

   ' Start listening for client requests.
   server.Start()

   ' Buffer for reading data
   Dim bytes(1024) As [Byte]
   Dim data As [String] = Nothing

   ' Enter the listening loop.
```



```
   While True
     Console.Write("Waiting for a connection on port 8080... ")

    ' Perform a blocking call to accept requests.

     Dim client As TcpClient = server.AcceptTcpClient()
     Console.WriteLine("Connected!")

     data = Nothing

     ' Get a stream object for reading and writing
     Dim stream As NetworkStream = client.GetStream()

     Dim i As Int32

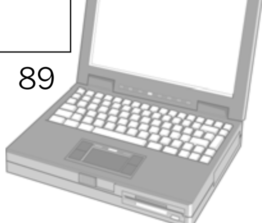
     ' Receive all the data sent by the client.
     i = stream.Read(bytes, 0, bytes.Length)
     If (i > 0) Then
       ' Translate data bytes to a ASCII string.
       data = System.Text.Encoding.ASCII.GetString(bytes, 0, i)
       Console.WriteLine([String].Format("Received: {0}", data))
     End If
     ' Serial Port sends message
     data = «<html><body> <h3> Checking room. Please, wait... </h3></
body></html>»
     Dim msg As Byte() = System.Text.Encoding.ASCII.GetBytes(data)
     stream.Write(msg, 0, msg.Length)

     recvMsg = "Nobody presents"
     SerialPort1.WriteLine("ALARM! Invasion!")
     System.Threading.Thread.Sleep(1000)

     data = "<html><body> <p> <h3>" & recvMsg & "</h3> </p></body></html>"
     msg = System.Text.Encoding.ASCII.GetBytes(data)
     stream.Write(msg, 0, msg.Length)

     stream.Close()

     ' Shutdown and end connection
     client.Close()
  End While
  Catch e As SocketException
      Console.WriteLine("SocketException: {0}", e)
End Try
SerialPort1.Close()
End Sub
    Sub SerialPort1_DataReceived(ByVal obj As Object, ByVal e As
System.IO.Ports.SerialDataReceivedEventArgs)
      recvMsg = SerialPort1.ReadLine()
    End Sub
End Module
```



Веб-сервер реализован с использованием блокирующих сокетов. В данном случае создается прослушивающий сокет `server`, который работает на локальной машине (сетевой адрес 127.0.0.1) и «прослушивает» запросы на соединение по порту 8080 в цикле `while`. При поступлении нового запроса создается клиентское соединение посредством рабочего сокета `client` с помощью оператора

```
Dim client As TcpClient = server.AcceptTcpClient()
```

Затем для чтения/записи данных через клиентский сокет создается объект «сетевого» потока `stream`:

```
Dim stream As NetworkStream = client.GetStream()
```

Поток `stream` инкапсулирует все операции чтения/записи, которые обычно реализуются функциями `recv` и `send` в языке С.

Для контроля сериального порта в программе создается экземпляр класса **SerialPort** и устанавливаются параметры обмена данными. Как и в первом проекте, состояние шлейфов TxD и RxD фиксируется обработчиком события **SerialPort1_DataReceived**.

При поступлении запроса от клиента через веб-браузер сервер отправляет данные по шлейфу от TxD к RxD:

```
SerialPort1.WriteLine("ALARM! Invasion!")
```

Если шлейф оказывается замкнутым, то последовальный порт принимает строку байтов и записывает ее в строку `recvMsg` (обработчик **SerialPort1_DataReceived**):

```
recvMsg = SerialPort1.ReadLine()
```

Таким образом, через 1 с после пересылки сообщения переменная `recvMsg` будет содержать одну из двух строк:

- «Nobody presents» – если шлейф разорван;
- «ALARM! Invasion!» – если шлейф замкнут, т. е. датчик сработал, и в помещении кто-то находится.

Эта строка отправляется по сети пользователю, сделавшему запрос. Таким образом, эта простейшая система позволяет контролировать безопасность помещений по Интернету. Естественно, что компьютер, на котором установлена подобная система, должен быть подключен к сети Интернет и быть соответствующим образом настроен. Откомпилируем проект и сохраним под именем webserver1.exe.

При запуске веб-сервера окно программы будет выглядеть так, как показано на рис. 4.15.

Теперь из любого стандартного браузера (в данном случае, используется Internet Explorer 6) попробуем получить данные о работе датчика перемещения через сервер. Напомню, что в нашей тестовой системе сервер работает с локальным IP-адресом 127.0.0.1 и получает запрос от браузера, запущенного на этой же машине. В общем случае, при соответствующих настройках сети данные сервера можно получить через браузер, запущенный на удаленной машине.

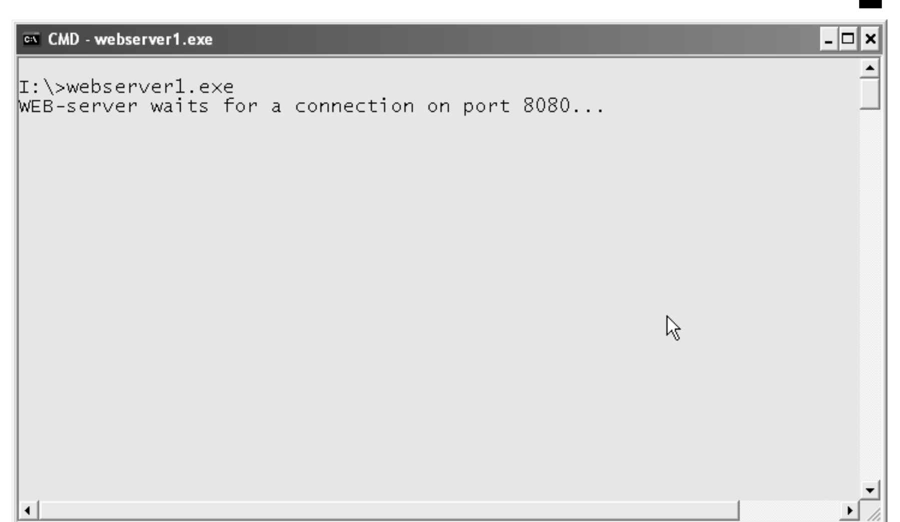


Рис. 4.15
Вид окна при запуске веб-сервера

Вот окно браузера, выполняющего запрос к нашему тестовому веб-серверу. Обратите внимание, как выглядит адресная строка (рис. 4.16).

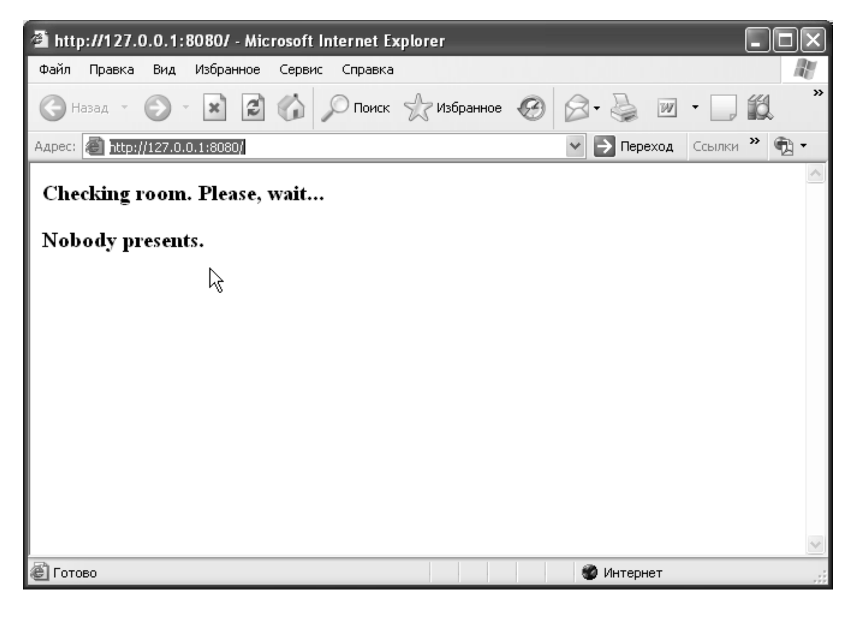


Рис. 4.16
Ответ пользователю при отсутствии срабатывания датчика перемещения

В адресной строке браузера вы должны ввести строку наподобие

http://127.0.0.1:8080

Как видим, в данном случае датчик не сработал и никакого перемещения в помещении нет. О принятии запроса от браузера можно судить по тексту, отображаемому в окне работающего сервера (рис. 4.17).

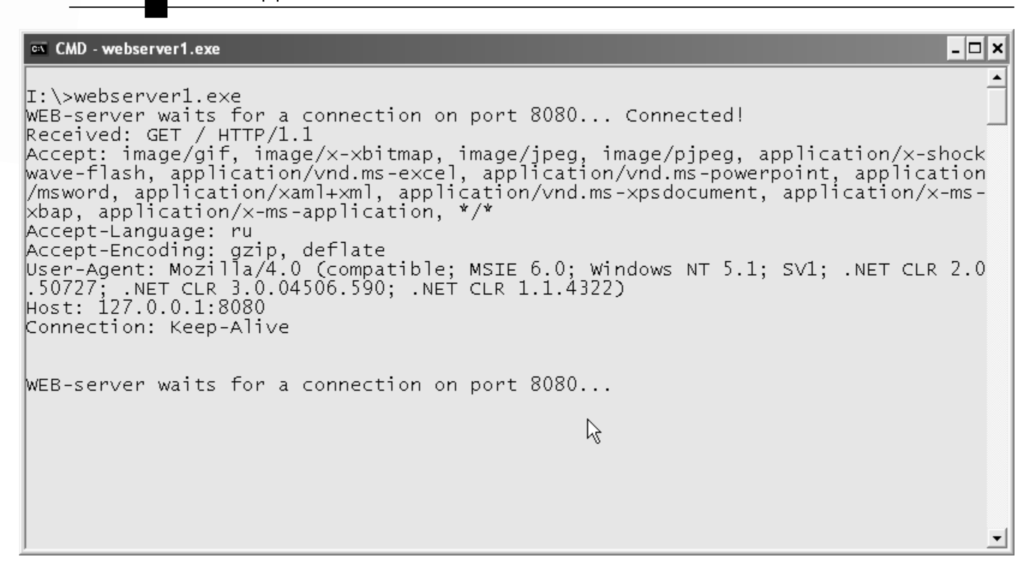


Рис. 4.17
Вид запроса браузера к нашему веб-серверу

Если в момент проверки имеется какое-либо перемещение, то ответ сервера будет выглядеть так, как показано на рис. 4.18.

Естественно, что данный проект можно значительно улучшить, если в аппаратной схеме устройства, например, фиксировать срабатывание датчика в регистре-защелке. Тогда схема будет сохранять это состояние сколь угодно долго, а не в течение короткого промежутка времени.

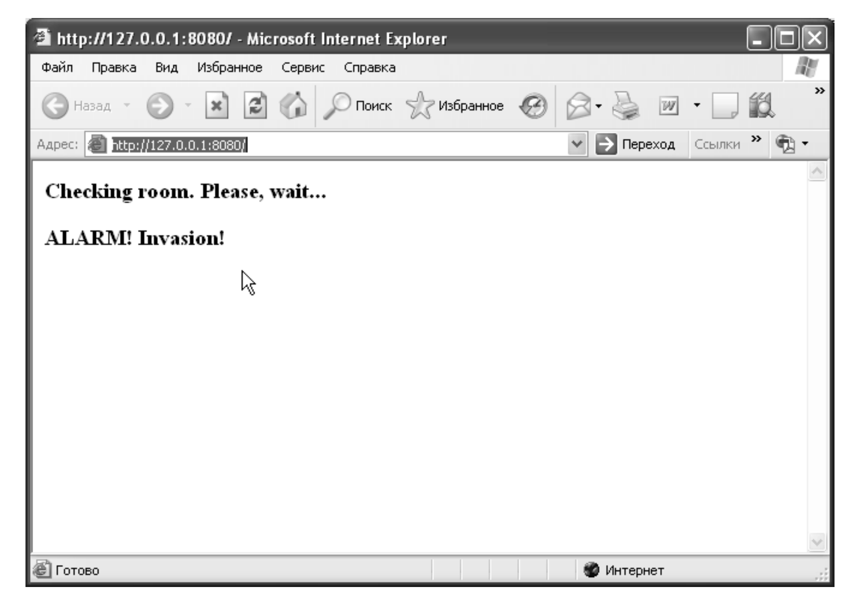
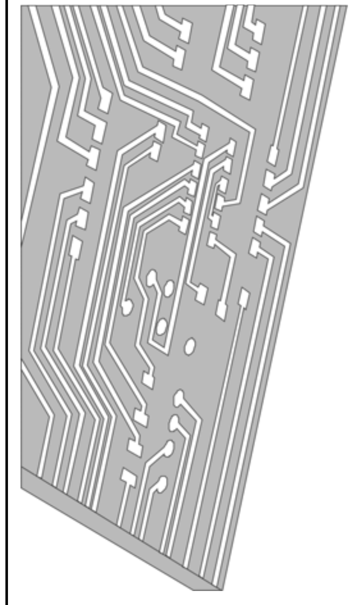
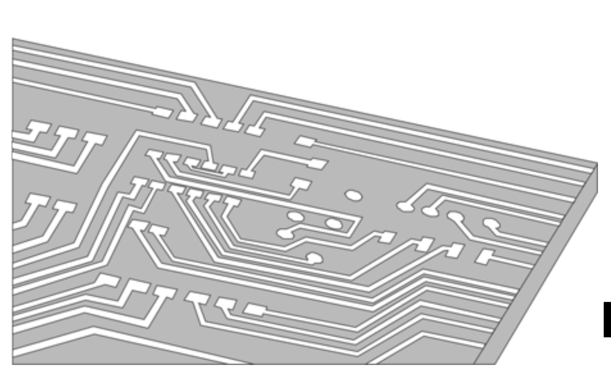


Рис. 4.18
Вид окна браузера при срабатывании датчика перемещения



Звуковые карты и их применение



5 Звуковые карты и их применение

Все современные персональные компьютеры позволяют эффективно обрабатывать звуковые (аудио) сигналы, включая проигрывание звуковых файлов, генерацию звуковых эффектов в компьютерных играх, запись и анализ аудиосигналов. Большинство пользователей знакомо с основными сферами применения звуковых карт. Тем не менее, звуковые карты находят широкое применение и в домашней лаборатории радиолюбителя, причем на их основе можно создавать довольно серьезные измерительные схемы, генераторы сигналов различной формы и схемы управления различными устройствами.

Применение звуковой карты в качестве «электронного осциллографа» известно широкому кругу читателей, причем разработан целый ряд программ, позволяющих запустить на вашем компьютере такой осциллограф. Несмотря на относительно узкий диапазон измерений, ограниченный верхним пределом звуковых частот (приблизительно 20 КГц), в большинстве случаев этого вполне достаточно для выполнения измерений относительно медленно изменяющихся аналоговых сигналов.

Перед тем как обсудить возможности работы со звуковыми картами, необходимо хотя бы вкратце ознакомиться с принципами обработки звуковых колебаний и их преобразования в цифровой сигнал. В основе работы звуковых карт, впрочем, как и многих других устройств, лежит принцип импульсно-кодовой модуляции (ИКМ, от англ. Pulse Code Modulation, PCM), который мы вкратце рассмотрим.

5.1. Импульсно-кодовая модуляция

Принцип импульсно-кодовой модуляции (ИКМ) проиллюстрирован на рис. 5.1.

При ИКМ из исходного сигнала F_S с определенной частотой F_{SAMPLE}, которая называется частотой дискретизации, выбираются значения (выборки), которые затем преобразуются в двоичный код. Если выборку представляют в виде 8-битового двоичного кода, то максимальное количество кодировок сигнала равно 256, если, например, выборка кодируется 12 битами, то максимальное количество выборок равно 2^{12} или 4096. Естественно, чем выше разрядность, тем точнее будут представлены дискретные значения сигнала.

Вторым важнейшим фактором, определяющим качество преобразованного сигнала, является частота дискретизации F_{SAMPLE}. При проектировании систем с ИКМ, да и с другими видами модуляции, основополагающим критерием является теорема Найквиста.

Применительно к принципу ИКМ ее можно сформулировать следующим образом: для получения однозначного соответствия цифрового кода аналоговому входному сигнала необ-

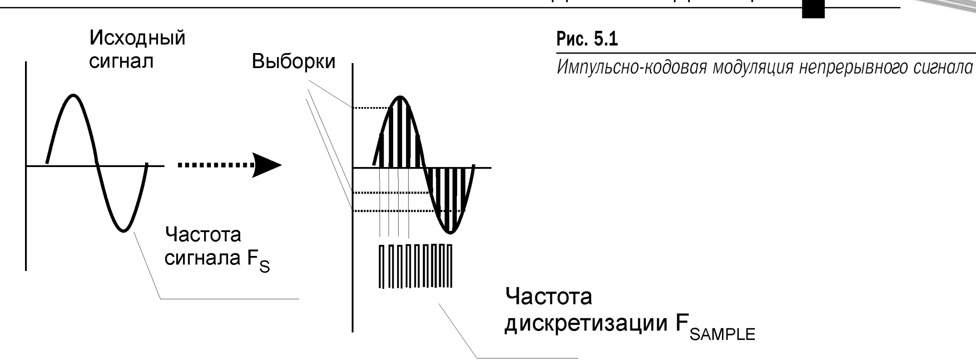


Рис. 5.1

Импульсно-кодовая модуляция непрерывного сигнала

ходимо, чтобы частота дискретизации, или, по-другому, частота квантования (F_{SAMPLE} в нашем примере) была бы как минимум в два раза выше максимальной частоты сигнала F_S.

Так, если верхнее значение частоты в спектре частот звукового сигнала равно 4 КГц, то частота квантования должна как минимум быть равной 8 КГц. Здесь важным является следующее замечание: критерий Найквиста гарантирует лишь однозначное соответствие дискретной выборки в определенный момент времени значению исходного сигнала в этот же момент времени, но не означает абсолютно точную передачу непрерывного сигнала во всем временном диапазоне. Речь здесь может идти о сколь угодно точном приближении цифрового образа к исходному сигналу.

Интуитивно очевидно, что чем выше частота квантования, тем точнее будет представление аналогового сигнала. Если, например, частота дискретизации будет равна 12 КГц, то количество выборок возрастет с 4000 до 12000, что обеспечит более точное цифровое представление исходного сигнала. В звуковых картах наивысшая частота диапазона обычно принимается равной 22 КГц, поэтому минимальная частота квантования равна как минимум 44 КГц.

То же самое касается и обратного преобразования цифрового кода в непрерывный аналоговый звуковой сигнал при воспроизведении звуковых файлов.

Метод импульсно-кодовой модуляции с некоторыми дополнениями и улучшениями реализован в звуковых картах современных персональных компьютеров.

В общем виде блок-схему звуковой карты можно представить следующим образом (рис. 5.2).

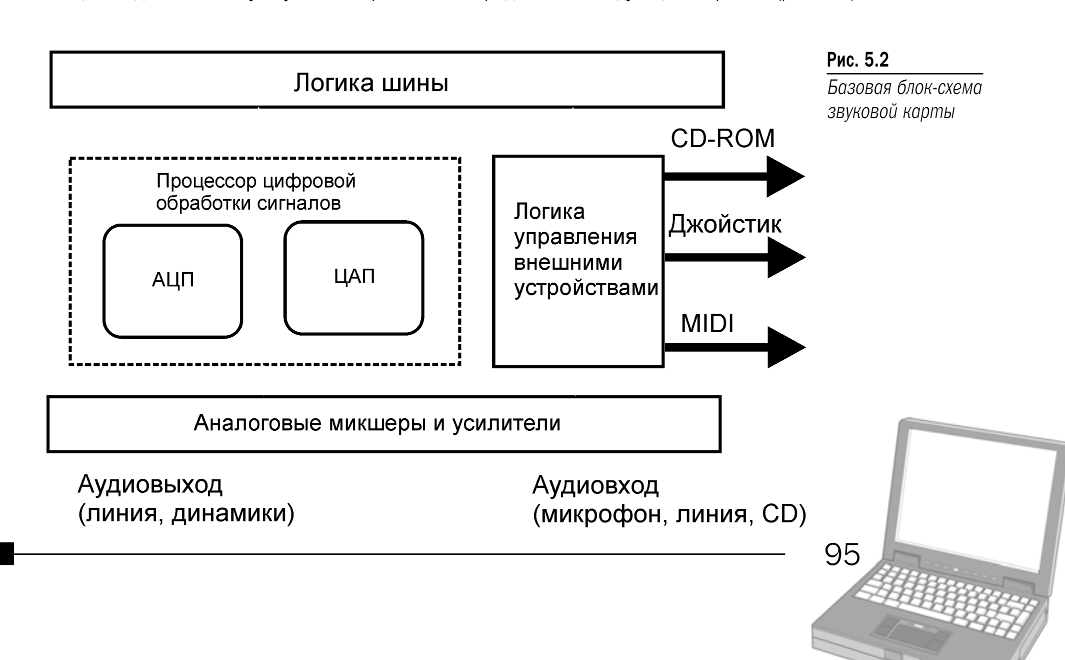
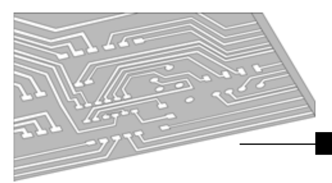


Рис. 5.2

Базовая блок-схема звуковой карты



Это довольно упрощенная функциональная схема, поскольку современные звуковые карты, особенно профессиональные, намного сложнее и могут содержать дополнительные узлы обработки и синтеза аудиосигналов. Тем не менее, базовая схема позволяет понять основные принципы обработки звука в таких устройствах.

В звуковой карте для преобразования аналогового звукового сигнала в цифровую форму и обратно применяется цифровой процессор, основными узлами которого являются аналого-цифровой (преобразование «сигнал-цифровой код») и цифро-аналоговый (преобразование «цифровой код-звуковой сигнал») преобразователи. Для того чтобы обработать входной сигнал или обеспечить стандартный уровень выходного сигнала, служат предварительные усилители входного сигнала и буферные усилители выходного сигнала. В этом же узле осуществляется предварительная фильтрация входных сигналов с помощью аналоговых фильтров нижних частот.

При практической реализации ИКМ в звуковых картах возникает проблема с так называемым «шумом квантования», который является артефактом для данного метода преобразования. При повышении частоты квантования возрастает достоверность (точность) преобразования сигнала в цифровую форму, однако требуются более сложные методы фильтрации возросшего шума квантования (например, использование «сигма-дельта» аналого-цифровых преобразователей с несколькими каскадами «сигма-дельта» модуляторов).

5.2. Звуковая карта в домашней лаборатории

Звуковую карту, кроме ее прямого назначения, можно использовать и в радиолюбительской практике для создания электронных устройств, начиная от самых простых и заканчивая весьма сложными. В этом смысле звуковая карта является уникальным устройством, поскольку позволяет разработать устройства, не прибегая к написанию сложных драйверов, что зачастую невозможно сделать для других устройств.

Например, очень часто звуковая карта применяется в качестве недорогого «электронного осциллографа» довольно высокого качества, причем для программирования такого устройства нет необходимости писать драйвер устройства, достаточно использовать многочисленные программные средства (мультимедийные API-функции операционной системы Windows или, например, библиотечные функции DirecSound из пакета DirectX SDK).

Мы не будем здесь рассматривать принципы разработки и программирования «электронных осциллографов» – тема довольно сложная и требует обширных знаний аппаратной и программной части как цифровых процессоров звуковых карт, так и самого компьютера. В Интернете имеется много различных программ «электронных осциллографов», которые можно использовать для измерения аналоговых сигналов низкой частоты с довольно подробными описаниями их работы.

Звуковая карта может с успехом применяться в качестве генератора сигналов различной формы (синусоиды, меандра, пилообразной и треугольной формы и т. д.). Это весьма распространенное применение этого устройства, и на рынке присутствует целый ряд программ «тон-генераторов», предназначенных для генерации сигналов различных частот.

Одной из относительно недорогих программ, присутствующих на рынке программных продуктов, является, например, NCH Tone Generator, с помощью которой можно генерировать на выходе звуковой карты сигналы различной формы. На ее примере мы рассмотрим принципы работы таких программ, тем более что такие программные тон-генераторы можно успешно использовать и при разработке собственных проектов.

На рис. 5.3 показаны установки параметров программы для получения сигнала прямоугольной формы с частотой 1000 Гц.

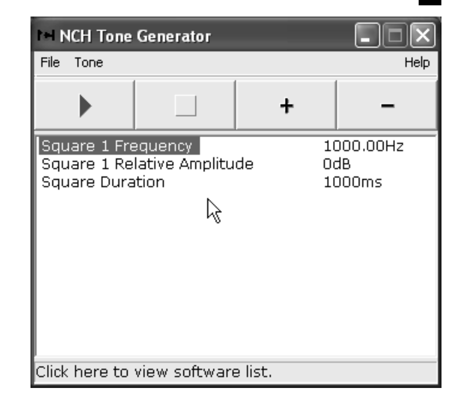


Рис. 5.3
Пример настройки входного сигнала прямоугольной формы

Из этого рисунка видно, что выбран прямоугольный сигнал (меандр) с частотой 1000 Гц и длительностью звучания 1000 мс (1 с). Для запуска генератора необходимо нажать левую кнопку ▶. В результате на линейном выходе звуковой карты в течение 1 с будет проигран прямоугольный сигнал с частотой 1000 Гц.

Для непрерывной генерации тона можно установить непрерывный режим проигрывания (рис. 5.4).

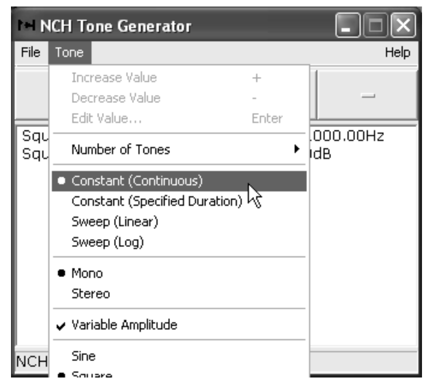


Рис. 5.4
Установка режима непрерывной генерации тона

Очень часто при настройке аппаратуры требуется подавать на устройство несколько сигналов разных частот и, возможно, разной формы, что может потребовать наличия нескольких выходов (звуковая карта имеет всего один линейный выход, неважно, это режим «стерео» или «моно»), а также переключения частот генератора. Мы рассмотрим варианты аппаратно-программной реализации такого многофазного генератора позже, а сейчас вернемся к описанию нашего тон-генератора.

Можно записать все требуемые типы сигналов с определенными частотами в отдельные звуковые файлы (обычно, это стандартные WAV-файлы) и затем проигрывать их в непрерывном режиме через любой мультимедиапроигрыватель. Все программы тон-генераторов позволяют записать тоновый сигнал определенной частоты в WAV-файл. В нашем примере запишем тоновый сигнал меандра в файл meandr.wav (рис. 5.5).

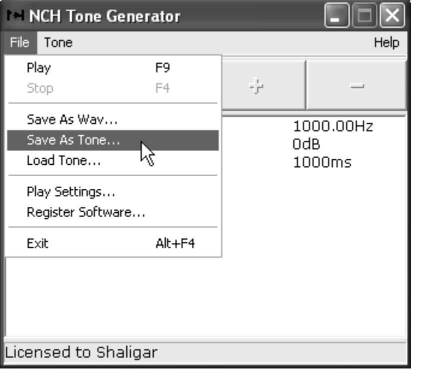
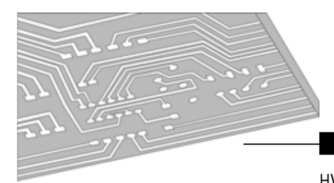


Рис. 5.5
Сохранение тонового сигнала в звуковом файле

Хотя можно использовать и такие генераторы сигналов, многие электронные схемы намного удобнее настраивать, используя небольшие специализированные приложения с возможностью быстрого выбора нескольких типов сигналов определенных частот. Кроме того, для схем несложных контроллеров, робототехнических устройств и схем бытовой автоматики очень удобно генерировать импульсные последовательности (пачки импульсов) определенной длительности через определенные интервалы времени, что зачастую невозможно сделать с помощью стандартных программ тон-генераторов и синтезаторов сигналов.

Можно написать довольно сложное приложение, которое сможет выполнять подобные функции, используя библиотеку стандартных мультимедийных функций Windows. Но можно пойти и другим путем. Для этого в любом тон-генераторе можно записать тоновые сигналы





нужной длительности в WAV-файлы, а затем проигрывать эти файлы в программе, написанной с использованием функций DirectSound из пакета DirectX SDK фирмы Microsoft. Функции DirectSound очень просты в использовании и позволяют создавать эффективные программы обработки звука буквально с помощью нескольких операторов. Следует учитывать и то, что функции этого пакета скрывают детали программирования звука, предлагая очень удобный и простой интерфейс для программирования. Что наиболее существенно, с функциями DirectSound можно работать из приложений, написанных на популярных языках программирования Visual Basic. NET и C# .NET. Единственное условие – пакет DirectX SDK должен быть установлен на вашем компьютере (его можно загрузить с сайта www.microsoft.com).

Чтобы продемонстрировать, насколько просто работать с DirectSound, напишем графическое приложение в Visual Basic .NET, позволяющее проиграть сохраненный ранее звуковой файл meandr.wav.

Для разработки приложения будем использовать среду Visual Studio .NET 2005 (можно использовать бесплатные версии Express Editions). Создадим каркас графического приложения в Visual Basic .NET, и на форме приложения поместим кнопку с текстом **Play WAV–file!**, при нажатии которой будет проигрываться файл meandr.wav (рис. 5.6).

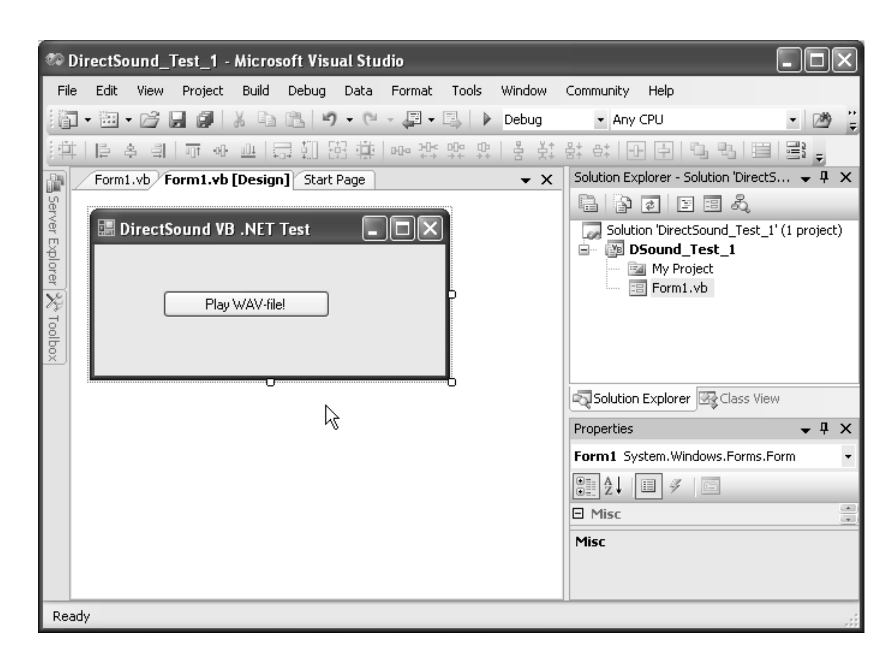


Рис. 5.6
Окно конструктора приложения

После этого следует обязательно добавить в приложение ссылки на пространство имен Microsoft.DirectX и Microsoft.DirectX.DirectSound (рис. 5.7, 5.8).

На закладке NET меню **Add Reference**, если DirectX SDK установлен, должны присутствовать соответствующие записи, ссылки на которые необходимо добавить в проект.

Затем следует импортировать пространства имен DirectX и DirectSound с помощью директив `Imports`, которые следует указать в начале исходного текста приложения:

```
Imports Microsoft.DirectX
Imports Microsoft.DirectX.DirectSound
```

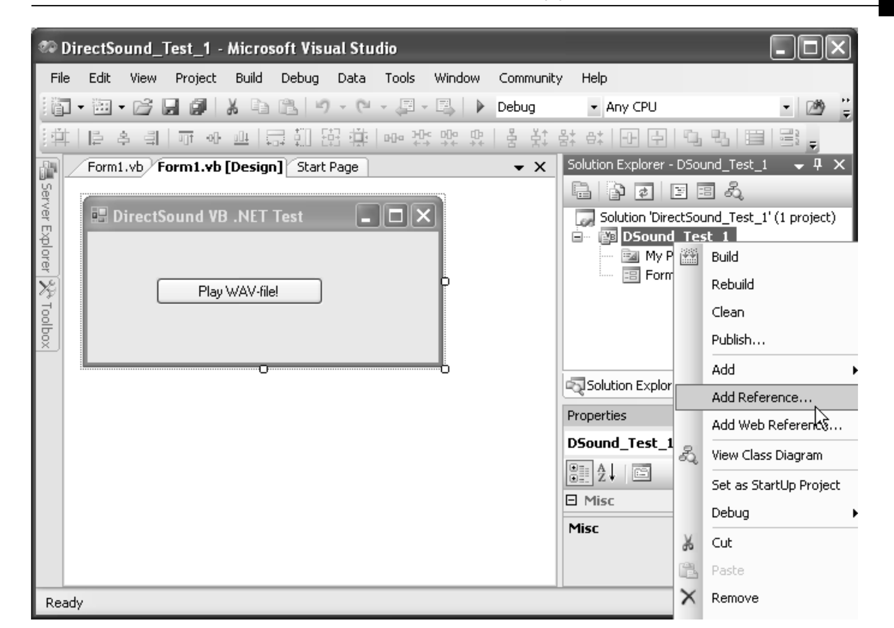


Рис. 5.7
Выбор опции меню для добавления ссылки на пространство имен

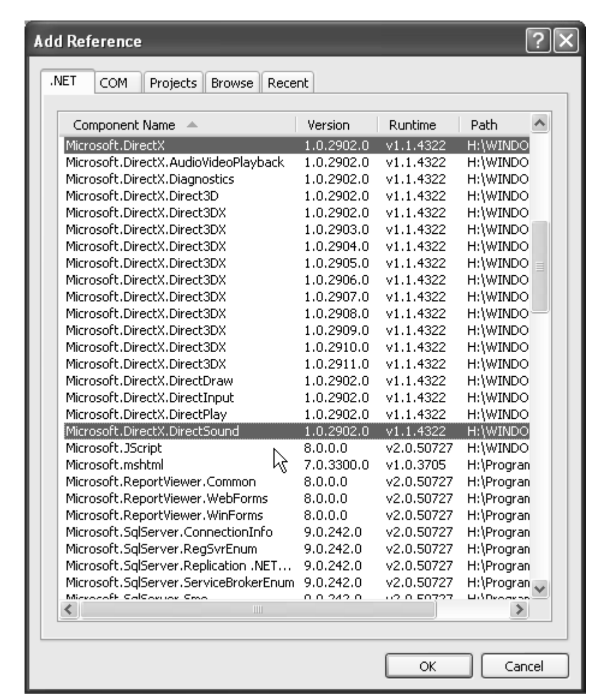
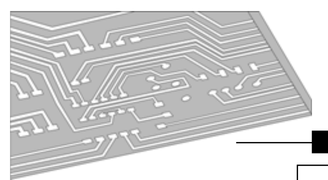


Рис. 5.8
Добавляемые ссылки на пространства имен

Для проигрывания звуковых данных следует определить логическое устройство (device), которое будет обрабатывать поток звуковых данных, и буфер потока, в котором будут находиться данные, после чего проиграть файл.

Вот полный исходный текст приложения:





```
Imports Microsoft.DirectX
Imports Microsoft.DirectX.DirectSound
Imports System.Windows.Forms

Public Class Form1
    Public device As Device
    Public buf As Microsoft.DirectX.DirectSound.Buffer

  Private Sub Button1_Click(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Button1.Click
    device = New Microsoft.DirectX.DirectSound.Device
    device.SetCooperativeLevel(Me, CooperativeLevel.Priority)
    buf = New Microsoft.DirectX.DirectSound.Buffer("i:\meandr.wav",
device)
    buf.Play(0, BufferPlayFlags.Default)
  End Sub
End Class
```

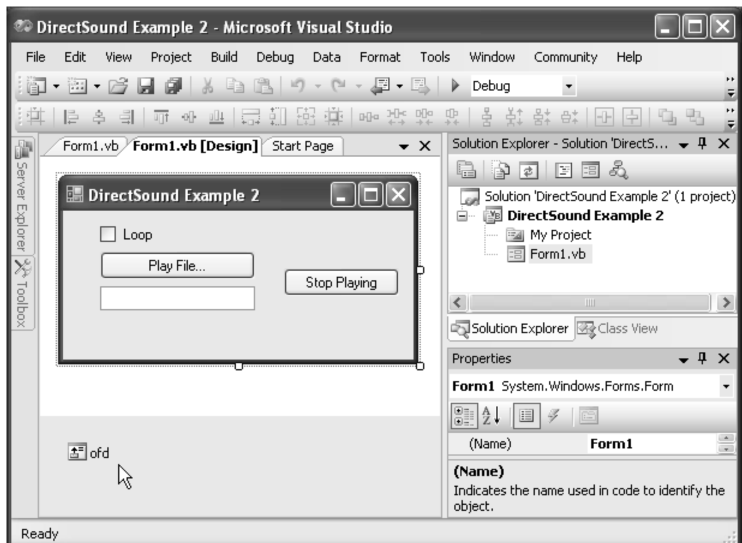
Как видно из листинга, программа получилась очень компактной. Единственное, что необходимо запомнить – это последовательность операций:

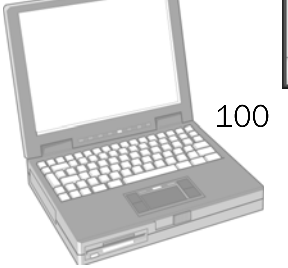
1. Создать контекст логического устройства воспроизведения (`device`).
2. Установить для этого устройства уровень доступа (`SetCooperativeLevel`).
3. Инициализировать потоковый буфер, связанный с данным устройством, и установить его параметры.
4. Проиграть звуковые данные с помощью метода `Play`.

Это простое приложение можно немного усовершенствовать, тогда можно будет проигрывать любые звуковые файлы, предварительно выбрав их в окне диалога и задав режим проигрывания (однократный или непрерывный). Ниже представлено окно конструктора приложения, разработанного в Visual Basic .NET (рис. 5.9).

Рис. 5.9

Вид окна конструктора приложения





Разместим на форме приложения дополнительные компоненты: CheckBox (для установки непрерывного режима проигрывания), OpenFileDialog (для выбора звукового файла) и окно текстового редактора, в котором будет отображаться путевое имя выбранного файла. Дополнительно для остановки проигрывания поместим на форму кнопку **Stop Playing**.

Приложение работает следующим образом: перед выбором звукового файла необходимо выбрать режим его проигрывания. Если установить отметку **Loop**, то файл будет проигрываться непрерывно, если оставить поле пустым – однократно.

Исходный текст по сравнению с предыдущим примером немного усложнился, но его можно понять без особого труда:

```
Imports Microsoft.DirectX
Imports Microsoft.DirectX.DirectSound

Public Class Form1
 Public device As Microsoft.DirectX.DirectSound.Device
 Public buf As Microsoft.DirectX.DirectSound.Buffer

 Public _loop As Boolean


Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
  device = New Microsoft.DirectX.DirectSound.Device
  device.SetCooperativeLevel(Me, CooperativeLevel.Priority)
  If ofd.ShowDialog() Then
   buf = New Microsoft.DirectX.DirectSound.Buffer(ofd.FileName, device)
   TextBox1.Text = ofd.FileName
   buf.Play(0, IIf(_loop, BufferPlayFlags.Looping,
BufferPlayFlags.Default))
  End If
    End Sub

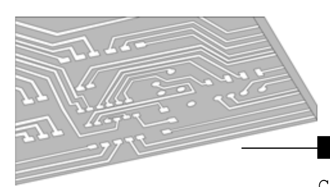
Private Sub Form1_Load(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Load
 _loop = False
End Sub

Private Sub CheckBox1_CheckedChanged(ByVal sender As System.Object, ByVal
e As System.EventArgs) Handles CheckBox1.CheckedChanged
 _loop = CheckBox1.Checked
End Sub

Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button2.Click
 buf.Stop()
End Sub
End Class
```

Здесь переменная _loop используется для определения режима проигрывания, принимая значения свойства Checked компонента CheckBox1 в обработчике CheckBox1_Che-





ckedChanged. Значение этой переменной используется в методе Play при проигрывании файла. Конструкция

```
IIf(_loop, BufferPlayFlags.Looping, BufferPlayFlags.Default)
```

позволяет выбрать, в зависимости от значения переменной _loop, режим пригрывания звуковых данных во вторичном буфере buf. Если значение _loop равно false, то буфер проигрывается однократно (выбирается флаг BufferPlayFlags.Default), если же значение этой переменно равно true, то будет выполняться непрерывное проигрывание (выбирается флаг BufferPlayFlags.Looping).

В остальном исходный текст приложения схож с предыдущим примером, а работу компонента OpenFileDialog, думаю, дополнительно объяснять нет необходимости.

Следующий пример, который мы рассмотрим, предоставляет широкие возможности для управления генерацией сигналов, позволяя устанавливать интервал времени, через который звуковой файл будет проигрываться. Если такой звуковой файл содержит, к примеру, 10 импульсов на частоте 1000 Гц, то установив паузу, например, в 1с, получим последовательность пачек импульсов, которые можно использовать при настройке робототехнических систем, систем инфракрасного и радиоуправления, в измерительных устройствах и т. д.

Для задания точных интервалов времени можно воспользоваться либо компонентом таймера их панели **ToolBox**, либо выбрать более эффективное решение – создать отдельный поток, который будет просыпаться через заданный интервал времени и запускать проигрывание звукового файла.

На рис. 5.10 показано окно конструктора нашего приложения.

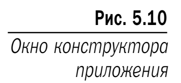
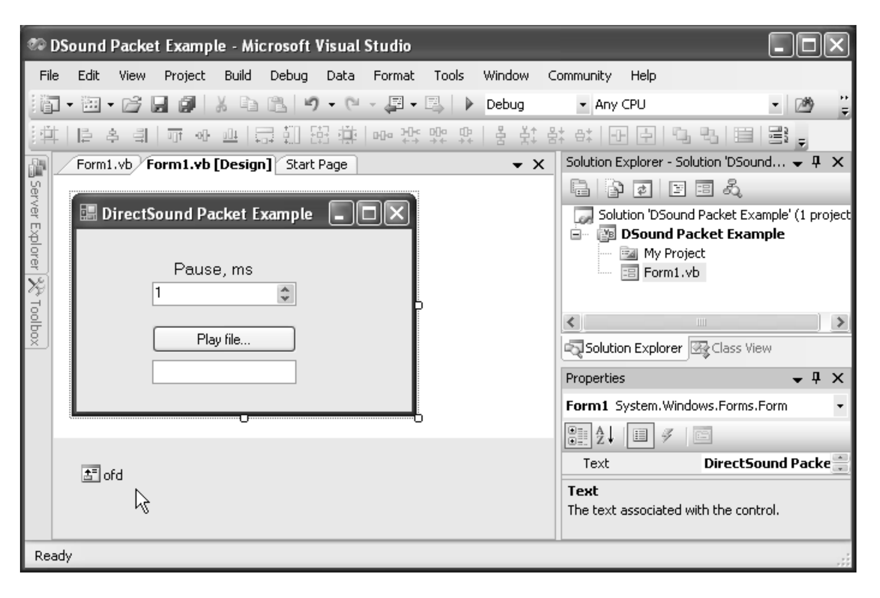


Рис. 5.10

Окно конструктора приложения



В этом проекте появился новый компонент NumericUpDown, позволяющий выбрать значение паузы между проигрыванием звукового файла. Зададим минимальное значение компонента равным 1, а максимальное – 32000. Свойство Value этого компонента будет



содержать текущее значение задержки в миллисекундах из диапазона 1–32000 и будет использоваться потоковой функцией для формирования паузы между пачками импульсов.

Вот исходный текст приложения:

```
Imports Microsoft.DirectX
Imports Microsoft.DirectX.DirectSound
Imports System.Threading

Public Class Form1
 Public device As Microsoft.DirectX.DirectSound.Device
 Public buf As Microsoft.DirectX.DirectSound.Buffer
 Public sthread As Thread
 Public tStart As Threading.ThreadStart
 Public delay As Integer

 Sub ThreadProc()
    While (True)
      buf.Play(0, BufferPlayFlags.Default)
      Thread.Sleep(delay)
    End While
 End Sub

Private Sub Form1_FormClosed(ByVal sender As Object, ByVal e As
System.Windows.Forms.FormClosedEventArgs) Handles Me.FormClosed
 sthread.Abort()
End Sub

Private Sub Form1_Load(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Load
  delay = 0
  device = New Microsoft.DirectX.DirectSound.Device
  device.SetCooperativeLevel(Me, CooperativeLevel.Priority)
  '
  tStart = New ThreadStart(AddressOf Me.ThreadProc)
  sthread = New Thread(tStart)
End Sub

Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
 If ofd.ShowDialog Then
     TextBox1.Text = ofd.FileName
     buf = New Microsoft.DirectX.DirectSound.Buffer(ofd.FileName, device)
     delay = UpDown1.Value
     sthread.Start()
 End If
End Sub
End Class
```

В этом приложении при его запуске (событие `Form1_Load`) создается и запускается отдельный поток `sthread`, функция `ThreadProc` которого проигрывает тоновый сигнал, а затем засыпает на время `delay`, которое определяется свойством `UpDown1.Value`.

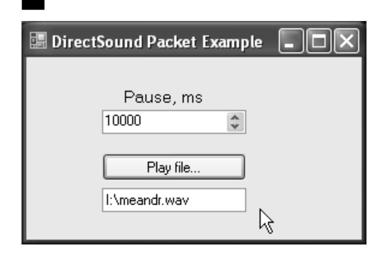


Рис. 5.11
Вид окна работающего приложения

При завершении работы программы работающий поток должен быть уничтожен, что и выполняется в обработчике события `Form1_FormClosed` с помощью метода `Abort` потока `sthread`.

Окно работающего приложения может выглядеть следующим образом (рис. 5.11).

Как видно из приведенных примеров, создание приложений для работы со звуковыми файлами с помощью библиотеки функций DirectSound не так и сложно.

Есть еще более простой способ, хотя и намного менее гибкий способ генерации звука в системе Windows. Он основан на использовании функции `Beep`. Если в среде Microsoft Visual Studio вы работаете с любым языком программирования, кроме Visual C++ .NET, то функция `Beep` указывается без аргументов и работает на частоте 1000 Гц, при этом звук посылается только на встроенный динамик компьютера.

Чуть больше возможностей имеет функция Beep в Visual C++ .NET. Здесь допускается задание двух аргументов: частоты и длительности звучания, хотя сигнал посылается только на встроенный динамик компьютера и имеет прямоугольную форму.

Тем не менее, если понаблюдать за линейным выходом звуковой карты с помощью осциллографа, то можно обнаружить, что сигнал, генерируемый функцией `Beep`, все же здесь присутствует, хотя его уровень и меньше, чем требуется стандартом. В принципе, такой сигнал можно использовать для настройки и управления другим оборудованием, если его усилить или, например, сформировать из него прямоугольный сигнал, совместимый по уровню с TTL-логикой.

Простая программа, написанная на Visual C++ .NET, демонстрирует, как можно задавать частоту и длительность звучания последовательности прямоугольных импульсов. Вот ее исходный текст:

```
#include <stdio.h>
#include <windows.h>

void main(int argc, char *argv[])
{
 int freq = 1000;
 int duration = 1000;

 if (argc != 3)
  {
   printf("Usage: %s freq duration\n", argv[0]);
   Beep(freq, duration);
   return;
  }

 freq = atoi(argv[1]);
 duration  = atoi(argv[2]);
 Beep(freq, duration);
 return;
}
```

При запуске этого приложения частоту и длительность следует задавать в качестве аргументов командной строки. Если аргументы не заданы, то генерируется звуковой сигнал частотой 1000 Гц в течение 1 с.

Мы рассмотрели некоторые программные принципы генерации звука с помощью звуковой карты. Тоновые последовательности, помимо их использования в качестве источников звуковых сигналов, можно применить и для других целей, но это требует подключения электронных устройств к линейному входу/выходу звуковой карты. Это довольно простые устройства, но они позволяют значительно расширить возможности применения звуковой карты. Рассмотрим некоторые из них.

5.3. Электронные устройства для работы со звуковой картой

Рассмотренные нами примеры программ работают со стандартным линейным выходом звуковой карты. Уровень выходного напряжения звуковой карты может достигать 2 В при выходном сопротивлении от десятков до сотен Ом. Этот уровень сигнала рассчитан на работу с аудиоаппаратурой различных производителей, причем подключение этой аппаратуры осуществляется стандартным кабелем (рис. 5.12).

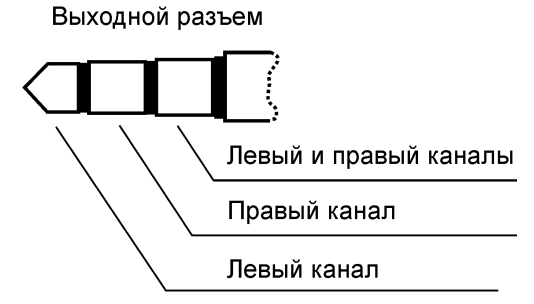


Рис. 5.12
Схема стандартного соединительного кабеля для выходного разъема карты

При соединении собственного устройства можно использовать один из каналов и общий провод обоих каналов в качестве общего провода схемы.

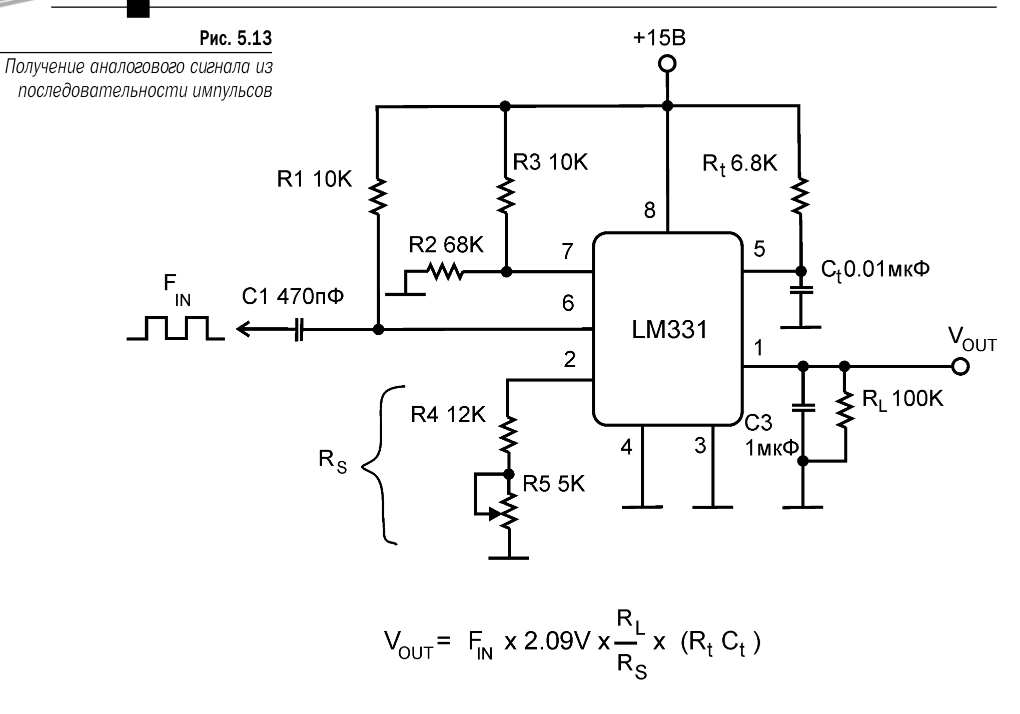
Сформированный программным способом, как показано ранее, сигнал можно применить в схеме цифро-аналогового преобразователя для формирования выходного аналогового напряжения, пропорционального частоте. Таким образом, ваша звуковая карта может стать источником регулируемого постоянного напряжения, которое можно использовать для управления различными устройствами и даже при разработке простого аналого-цифрового преобразователя.

Обычно преобразователи «частота-напряжение» требуют на входе прямоугольного сигнала, совместимого по уровню с TTL-логикой, поэтому следует сформировать такой сигнал с помощью компаратора, выполненного либо на специализированной микросхеме, либо на операционном усилителе. Вот один из вариантов получения аналогового выходного напряжения из последовательности импульсов (рис. 5.13).

На этом рисунке показана схема прецизионного цифро-аналогового преобразователя, реализованного по принципу преобразования «частота-напряжение», работающего в диапазоне входных частот от 0 до 10 КГц и реализованного на популярной микросхеме LM331

Рис. 5.13

Получение аналогового сигнала из последовательности импульсов



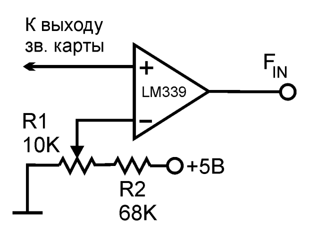
фирмы National Semiconductor. При указанных значениях компонентов формула для получения выходного напряжения упрощается:

$V_{OUT} = 0.001F_{IN}$.

Таким образом, при верхнем значении частоты в 10 КГц выходное напряжение преобразователя будет равно 1 В.

Рис. 5.14

Формирователь прямоугольного сигнала



Для надежной работы преобразователя сигнал на конденсатор C1 можно подавать посредством компаратора, который сформирует крутые фронты сигнала и обеспечит стабильный перепад амплитуд. Схема такой промежуточной цепи очень проста (рис. 5.14).

В этой схеме с помощью переменного резистора R1 следует установить напряжение на инвертирующем входе порядка сотни милливольт или меньше (в этом случае может понадобиться компаратор с низким уровнем напряжения смещения на входах). Чем ниже напряжение смещения на инвертирующем входе, тем точнее частота на выходе компаратора будет соответствовать частоте входного сигнала. Если же на неинвертирующий вход компаратора поступают прямоугольные импульсы, то особо беспокоиться о задании смещения не стоит – достаточно выбрать его меньшим наименьшей амплитуды напряжения входного сигнала.

Программное обеспечение для такого преобразователя создать очень просто. Пусть нам требуется получить на выходе преобразователя 10 фиксированных значений напряжения – от 0.1 до 1 В. В этом случае частота входного сигнала должна иметь фиксированные значения от 1000 до 10000 Гц с шагом 1000 Гц. Иными словами, для получения напряжения 0.1 В на выходе преобразователя на его вход необходимо подать через компаратор (рис. 5.14) сигнал

частотой 1000 Гц, для получения выходного напряжения 0.5 В на входе преобразователя должна присутствовать частота 5000 Гц и т. д.

Перед созданием такой программы запишем с помощью одной из программ тон-генераторов 10 звуковых файлов с тоновыми сигналами прямоугольной формы от 1000 до 10000 Гц и назовем их meandr1000.wav, meandr2000.wav и т. д. Затем в Visual Basic .NET разработаем приложение, окно конструктора которого будет выглядеть так, как показано на рис. 5.15.

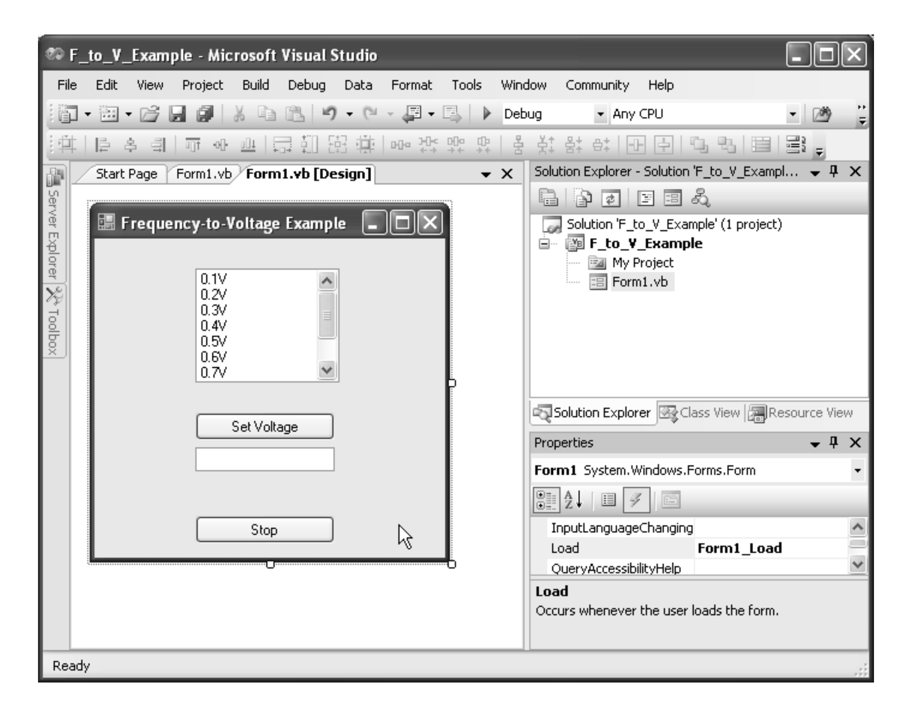


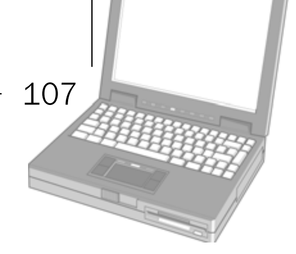
Рис. 5.15
Окно конструктора приложения

Поместим на нашу форму две кнопки (**Set Voltage** и **Stop**), однострочный текстовый редактор `TextBox` и список выбора `ListBox`. Программа работает следующим образом: из списка выбирается требуемый уровень выходного напряжения преобразователя, после чего следует нажать кнопку **Set Voltage**, обработчик которой проигрывает соответствующий выбранному напряжению звуковой файл. Для проигрывания файла в этом же обработчике запускается отдельный поток. Остановить проигрывание можно, нажав кнопку **Stop**, обработчик нажатия которой завершает поток.

Вот исходный текст приложения:

```
Imports Microsoft.DirectX
Imports Microsoft.DirectX.DirectSound
Imports System.Threading.Thread

Public Class Form1
    Public device As Microsoft.DirectX.DirectSound.Device
    Public buf As Microsoft.DirectX.DirectSound.Buffer
    Public threadStart As Threading.ThreadStart
```

```
    Public thread1 As Threading.Thread

    '
    Public file0 As String = "i:\meandr1000.wav"
    Public file1 As String = "i:\meandr2000.wav"
    Public file2 As String = "i:\meandr3000.wav"
    Public file3 As String = "i:\meandr4000.wav"
    Public file4 As String = "i:\meandr5000.wav"
    Public file5 As String = "i:\meandr6000.wav"
    Public file6 As String = "i:\meandr7000.wav"
    Public file7 As String = "i:\meandr8000.wav"
    Public file8 As String = "i:\meandr9000.wav"
    Public file9 As String = "i:\meandr10000.wav"

    Private Sub OnPlay_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles OnPlay.Click

        Dim listIndex As Integer = ListBox1.SelectedIndex
        TextBox1.Text = ListBox1.SelectedItem.ToString
        threadStart = New Threading.ThreadStart(AddressOf threadProc)
        thread1 = New Threading.Thread(threadStart)

        Select Case listIndex
         Case 0
          buf = New Microsoft.DirectX.DirectSound.Buffer(file0, device)
         Case 1
          buf = New Microsoft.DirectX.DirectSound.Buffer(file1, device)
         Case 2
          buf = New Microsoft.DirectX.DirectSound.Buffer(file2, device)
         Case 3
          buf = New Microsoft.DirectX.DirectSound.Buffer(file3, device)
         Case 4
          buf = New Microsoft.DirectX.DirectSound.Buffer(file4, device)
         Case 5
          buf = New Microsoft.DirectX.DirectSound.Buffer(file5, device)
         Case 6
          buf = New Microsoft.DirectX.DirectSound.Buffer(file6, device)
         Case 7
          buf = New Microsoft.DirectX.DirectSound.Buffer(file7, device)
         Case 8
          buf = New Microsoft.DirectX.DirectSound.Buffer(file8, device)
         Case 9
          buf = New Microsoft.DirectX.DirectSound.Buffer(file9, device)
        End Select
        OnPlay.Enabled = False
        OnStop.Enabled = True
        thread1.Start()
    End Sub

    Private Sub Form1_Load(ByVal sender As Object, ByVal e As
```

```
System.EventArgs) Handles Me.Load
        device = New Microsoft.DirectX.DirectSound.Device()
        device.SetCooperativeLevel(Me, CooperativeLevel.Priority)
        OnStop.Enabled = False
    End Sub

    Private Sub OnStop_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles OnStop.Click
        thread1.Abort()
        OnPlay.Enabled = True
        OnStop.Enabled = False
    End Sub

    Public Sub threadProc()
        While (True)
            buf.Play(0, BufferPlayFlags.Default)
        End While
    End Sub

    Private Sub Form1_FormClosed(ByVal sender As System.Object, ByVal e
As System.Windows.Forms.FormClosedEventArgs) Handles MyBase.FormClosed
        thread1.Abort()
    End Sub
End Class
```

Напомню, что при разработке приложения необходимо добавить ссылки на пространства имен Microsoft.DirecX и Microsoft.DirectX.DirectSound. Кроме этого, необходимо импортировать эти пространства имен с помощью директивы `Imports`.

Поскольку в этом приложении используется отдельный поток, вторичный буфер `buf` может проигрываться в обычном (не в циклическом!) режиме.

Для реализации программной части преобразователя «частота-напряжение» можно воспользоваться и функцией `Beep`, работу которой мы проанализировали ранее.

Получить постоянное напряжение переменной величины из звукового сигнала можно и другим способом, преобразовав среднее значение уровня выходного напряжения звуковой карты в постоянное напряжение на выходах преобразователя, реализованного на широко распространенной микросхеме индикатора уровня напряжения LM3915 в нестандартном включении. Правда, в этом случае диапазон выходных напряжений будет ограничен 10-ю значениями, но для многих приложений этого может оказаться вполне достаточно.

Схема преобразователя «амплитуда-напряжение» показана на рис. 5.16.

На транзисторе Q1 и D1 собран однополупериодный детектор. Выпрямленное напряжение будет пропорционально уровню входного сигнала. Выходной операционный усилитель DA2 включен по схеме суммирующего усилителя с инверсией.

На неинвертирующий вход операционного усилителя подается напряжение смещения Uп, которое должно быть больше напряжения питания микросхемы DA1 на 1-2 В. Если выбрать значения резисторов на выходах DA1 одинаковыми и равными R, то выходное напряжение на выходе операционного усилителя DA2 будет определяться формулой

$$U_{вых} = U_п \times (1 + nR_F/R) - (U_1 + U_2 + \ldots + U_n) \times (R_F/R),$$

где n – количество задействованных выходов микросхемы DA1.

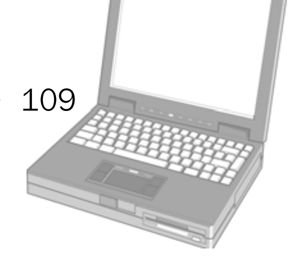
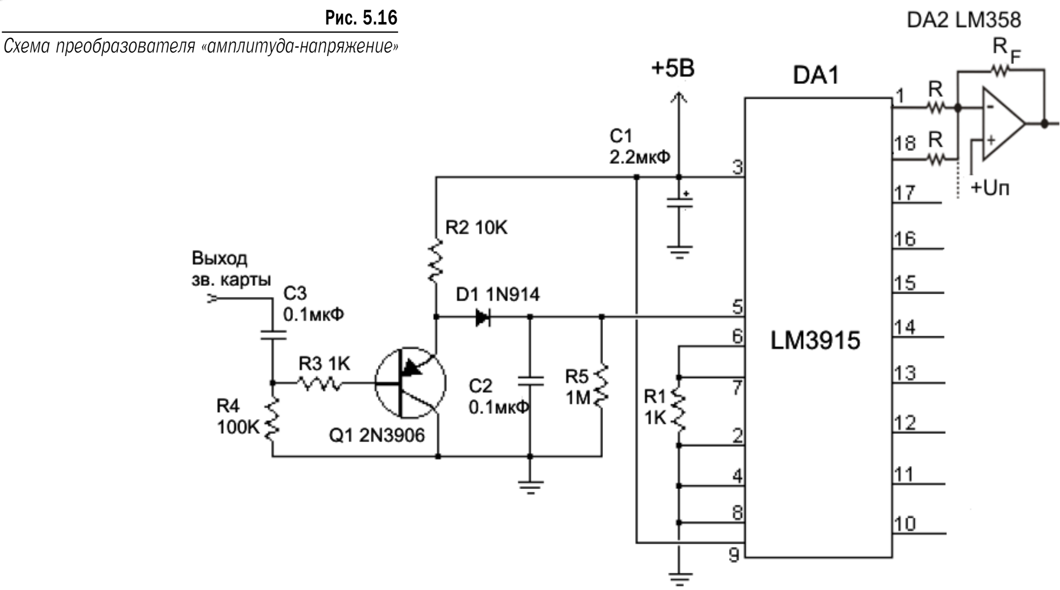


Рис. 5.16

Схема преобразователя «амплитуда-напряжение»



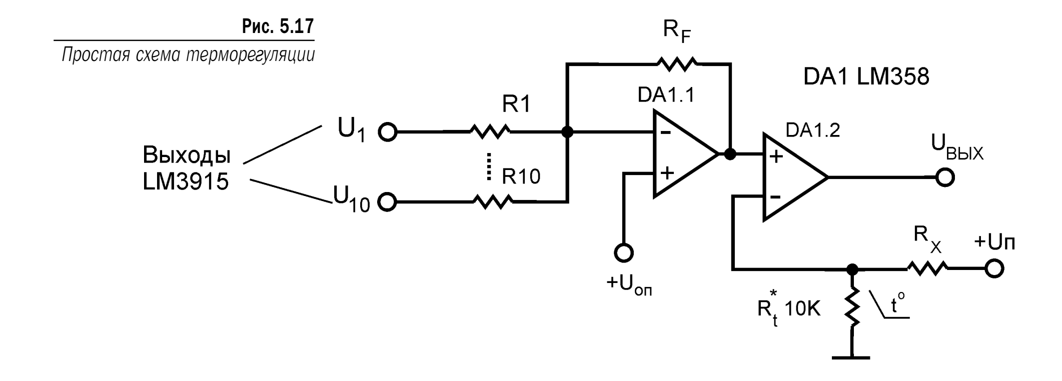
Исходя из этих соображений, напряжение питания операционного усилителя DA2 следует выбирать больше напряжения питания Uп. Значения резисторов R и R_F можно выбрать в пределах единиц – десятков ом.

К выходу операционного усилителя можно подключать нагрузку, например, электродвигатель или электроосвещение и управлять этой нагрузкой путем изменения амплитуды звукового сигнала.

На основе этой схемы можно разработать простую схему терморегуляции (рис. 5.17).

Рис. 5.17

Простая схема терморегуляции



В этой схеме в качестве терморегулирующего элемента выбран терморезистор с отрицательным температурным коэффициентом. При достижении определенной температуры объекта напряжение на выходе делителя R_X – R_t становится меньше напряжения на неинвертирующем входе компаратора DA1.2, в результате чего выход компаратора переключается в низ-

кий уровень, который может управлять какой-либо нагрузкой (например, электронагревательным устройством).

Данная схема позволяет выполнить настройки терморегулятора как в цепи делителя R_X – R_t, так и в цепи амплитудного детектора на микросхеме LM3915 путем подачи на вход схемы большей или меньшей амплитуды сигнала со звуковой карты. Все номиналы резисторов в этой, как и в предыдущей схеме, выбираются исходя из настроек вашей системы, и никакой сложности этот процесс не представляет.

Вместо микросхемы LM3915 можно с успехом использовать два учетверенных компаратора, например, LM324, установив на их входах (инвертирующих или неинвертирующих) соответствующие напряжения смещения, которые следует подбирать экспериментально. Если нет компаратора, то можно использовать операционные усилители общего назначения, например, LM358, LF451 и т. д.

Прецизионную схему терморегулятора можно построить на базе схемы, показанной на рис. 5.13 (если у вас имеется микросхема LM131/LM231/LM331). Она может обеспечить очень точную регулировку температуры, которая будет зависеть только от точности используемого термодатчика. Рассмотрим схему прецизионного терморегулятора (рис. 5.18).

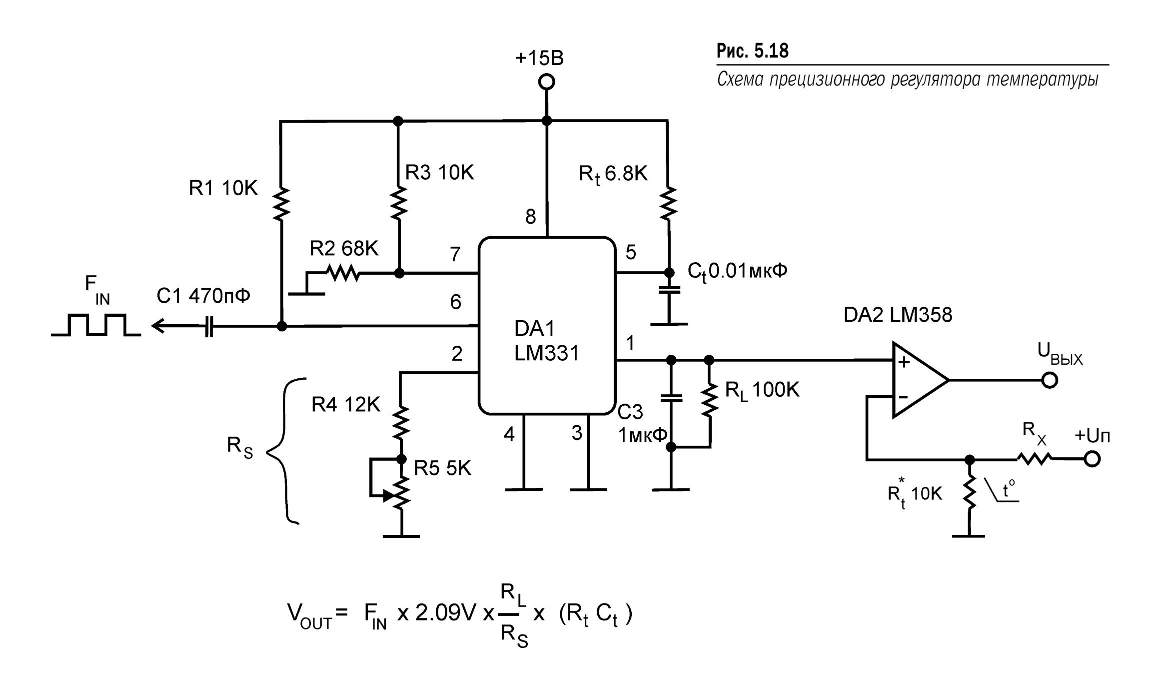
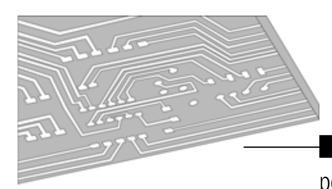


Рис. 5.18

Схема прецизионного регулятора температуры

Здесь напряжение на входе компаратора DA2 прямо пропорционально частоте сигнала на выходе звуковой карты, поэтому можно задавать регулируемую температуру объекта непосредственно из вашего приложения. В этом случае необходимо указать зависимость (по формуле или в виде таблицы) температуры от частоты. Это не очень сложно, учитывая то, что преобразователь «частота-напряжение» на микросхеме LM331 выполняет большую часть работы за вас, поэтому достаточно привязать выходное напряжение преобразователя к напряжению на делителе R_X–R_t, что труда не представляет. Можно еще более улучшить характеристики схемы, установив корректирующий резистор параллельно с терморезисто-





ром для линеаризации температурной характеристики, однако это потребует дополнительных расчетов.

Поскольку звуковая карта имеет стереовыход, то в показанных выше схемах можно получить максимум два управляющих напряжения для внешних устройств. Тем не менее, можно расширить количество выходных сигналов, управляемых с линейного выхода звуковой карты. Для этого потребуется создать чуть более сложную схему такого интерфейса. Разработаем устройство, позволяющее получить из одного и того же звукового сигнала 4 выхода управления.

Для этого воспользуемся схемотехническим решением, основанным на применении активных фильтров. Прежде чем вкратце описать суть применения активных фильтров, немного остановимся на разложении непрерывного периодического сигнала на гармонические составляющие, т. е. на синусоидальные и косинусоидальные составляющие. Этот метод называется разложением в ряд Фурье и основан на том, что сигнал прямоугольной формы содержит множество гармоник различной амплитуды, причем наибольшей амплитудой будет обладать гармоническая составляющая на частоте импульсов. Не углубляясь в теорию, вышеизложенное можно сформулировать так: любой непрерывный сложный сигнал несинусоидальной формы в общем случае включает бесконечное число гармонических составляющих, амплитуда и частота которых может быть определена при разложении такого сигнала в ряд Фурье.

Так, например, последовательность прямоугольных импульсов, следующих с частотой 1000 Гц, будет, в общем случае, содержать различные гармоники сигнала, но наибольший вклад будет вносить составляющая на частоте 1000 Гц. Не вдаваясь в детали, для импульсной последовательности f(x) единичной амплитуды разложение в ряд Фурье можно представить суммой нечетных гармоник синусоидальных составляющих вида

$$4/\pi \times [\sin(x)/1 + \sin(3x)/3 + \sin(5x)/5 + \ldots]$$

Из формулы легко увидеть, что наибольшее значение амплитуды, равное $4/\pi$, имеет основная гармоника сигнала, следующая (на частоте, в 3 раза выше основной) имеет амплитуду в 3 раза меньше и т. д.

Если, например, прямоугольные импульсные сигналы частотой 1000 Гц подать на вход активного узкополосного фильтра, настроенного для пропускания именно этой частоты, то на его выходе максимальную амплитуду будет иметь синусоидальный сигнал с частотой 1000 Гц, в то время как остальные составляющие будут значительно, или почти полностью, ослаблены. Иными словами, на выходе такого фильтра будет присутствовать синусоида частоты 1000 Гц, и, возможно, небольшие сигналы высших гармоник. Схема активного узкополосного фильтра для частоты 1000 Гц показана на рис. 5.19.

Это один из вариантов популярных фильтров типа Салена-Ки. Помимо того, что данная схема выделяет из импульсной последовательности синусоидальный сигнал частотой 1000 Гц, она и усиливает его в несколько раз (теоретический коэффициент усиления этой схемы при идеальном согласовании равен 10, но реальный может быть несколько меньше, к тому же влияние оказывает и скорость нарастания выходного сигнала конкретного операционного усилителя). Для расчета активных фильтров имеется много бесплатных программ ведущих фирм-производителей (Texas Instrument, Microchip и т. д.), с помощью которых, не вникая в довольно сложные математические расчеты, можно по заданным характеристикам определить значения резисторов и конденсаторов активного фильтра.

Полученный на выходе фильтра синусоидальный сигнал можно с помощью диодного амплитудного детектора преобразовать в сигнал постоянного тока и использовать для управления нагрузкой, например, включения/выключения электродвигателя постоянного тока. Пример такой схемы управления показан на рис. 5.20.

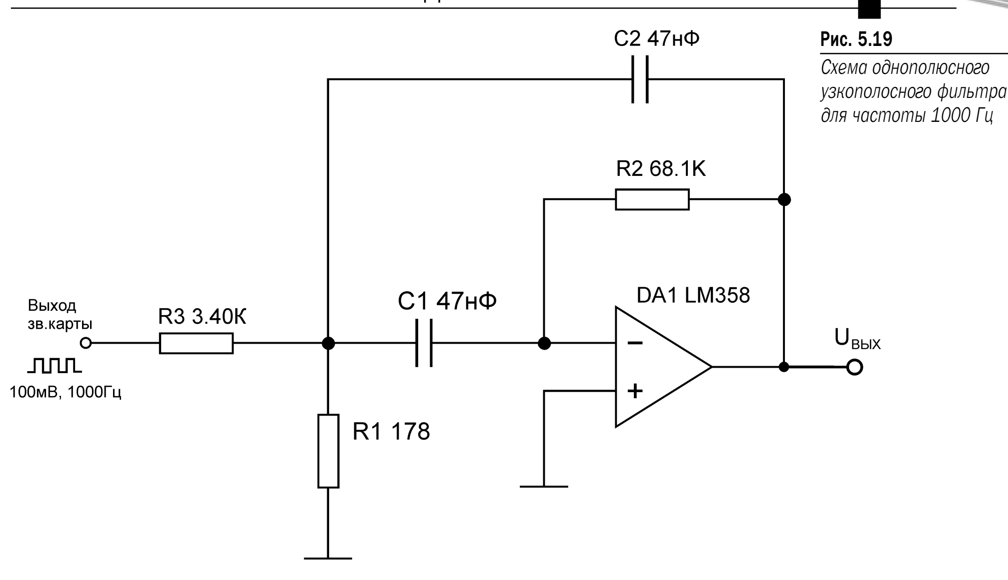


Рис. 5.19
Схема однополюсного узкополосного фильтра для частоты 1000 Гц

В данной схеме прямоугольный сигнал с частотой 1000 Гц и амплитудой около 100 мВ подается на вход полосового фильтра на микросхеме DA1 с полосой пропускания около 200 Гц. Синусоидальная составляющая с частотой 1000 Гц подается на вход амплитудного детектора на диоде D1. Выпрямленное напряжение попадает на неинвертирующий вход компаратора на микросхеме DA2, и, если превышает напряжение $U_{СМ}$ на инвертирующем входе, то вызывает переключение выхода в состояние лог.1. В результате мощный полевой транзистор Q1 открывается, замыкая цепь питания электродвигателя M1.

Если частота сигнала на выходе звуковой карты изменяется, например, к значению 1200 Гц, то амплитуда напряжения на выходе фильтра резко падает и соответственно уменьшается постоянное напряжение на неинвертирующем входе компаратора DA2. Компаратор переключается в низкий уровень, и транзистор Q1 закрывается, разрывая цепь двигателя M1. Напряжение смещения $U_{СМ}$ можно установить равным нескольким десяткам милливольт, при этом уровень звукового сигнала на выходе звуковой карты, при котором срабатывает схема, можно установить программно, непосредственно в регуляторе громкости звуковой карты, который обычно располагается на панели **Пуск** рабочего стола Windows.

По схожей методике можно разработать, например, полосовые фильтры на частоты 2000, 3000, 4000 Гц и так далее с соответствующими цепями амплитудного детектора и ключа на полевом транзисторе, увеличив тем самым количество выходов управления.

Подключив входы таких фильтров к выходу звуковой карты, можно обеспечить сигналы управления для нескольких независимых каналов. Если, например, с выхода звуковой карты подается сигнал 3000 Гц, то он будет пропущен только цепью фильтра, настроенного на частоту 3000 Гц, в то время как остальными фильтрами такой сигнал будет подавлен. Например, схема управления для 4-х независимых каналов может выглядеть так (рис. 5.21):

В этой схеме один выход звуковой карты используется для управления четырьмя цепями нагрузки, для чего можно воспользоваться одной из программ, рассмотренных нами ранее. Меняя частоту на выходе звуковой карты от 1000 до 4000 Гц, можно отдельно включать каждую цепь. Естественно, можно усовершенствовать разработку, добавив еще несколько каналов управления, например, для частот 5000, 6000 Гц и т. д. При этом необходимо будет



Рис. 5.20

Схема управления электродвигателем с помощью звуковой карты

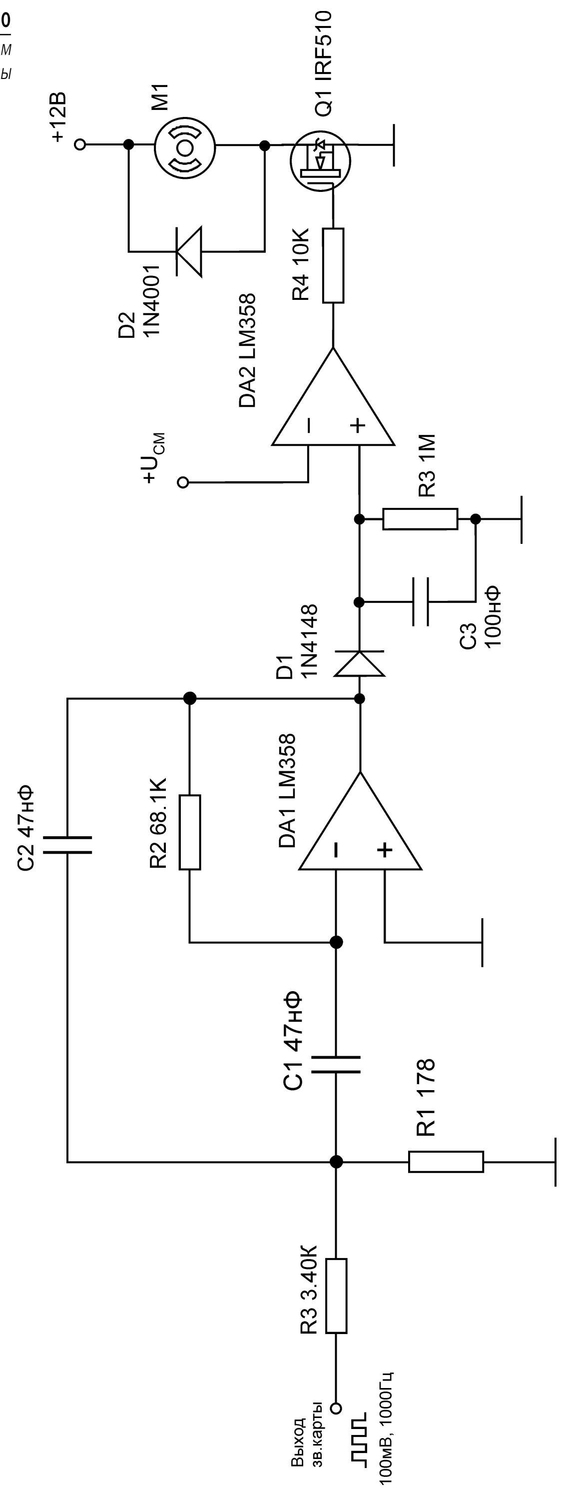




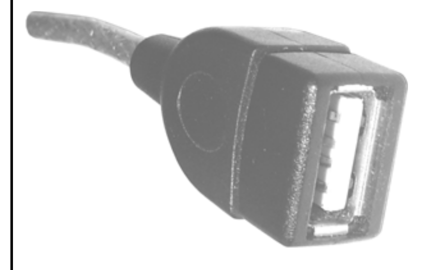
Рис. 5.21
Схема управления несколькими цепями с одного выхода звуковой карты

с помощью программ для расчета фильтров вычислить значения компонентов (резисторов и конденсаторов) для каждого полосового фильтра.

Если сделать фильтры еще более узкополосными, то интервал частот можно уплотнить и сделать шаг, равный, например, 300 Гц.

Возможностей применения звуковой карты намного больше, чем те, которые были нами рассмотрены. Так, например, чувствительность микрофонного звукового входа можно значительно повысить, установив цепь предварительного микрофонного усилителя, что позволит работать с любыми типами микрофонов. Подобные схемы очень часто встречаются в Интернете, причем большинство из них может легко собрать радиолюбитель средней руки.

Входы звуковой карты можно использовать и для подключения различных схем датчиков, например, температуры и вибрации. Это более сложные аспекты применения звуковых карт, которые здесь мы описывать не будем; некоторые такие проекты встречаются в Интернете.





Интерфейсы USB и Bluetooth



6 Интерфейсы USB и Bluetooth

Эта глава посвящена анализу архитектурных и программных особенностей интерфейсов USB и Bluetooth и их использованию в домашней лаборатории. Обсуждение начнем с интерфейса USB. Универсальная последовательная шина обмена данными (Universal Serial Bus, USB) является широко известным стандартом последовательной передачи данных для создания интерфейсов периферийных устройств с компьютерными системами. Шина USB разработана, чтобы служить расширением промышленного стандарта к архитектуре компьютеров. Широко применяется также и беспроводная реализация универсальной последовательной шины, известная под названием Wireless USB.

Изначально USB была задумана как универсальный стандарт, позволяющий использовать унифицированную методику для подключения периферийных устройств к компьютеру, заменив последовательный и параллельный интерфейсы. USB имеет целый ряд преимуществ по сравнению с этими интерфейсами:

Каковы же преимущества USB-шины и как их можно использовать? Можно выделить несколько основных характеристик шины:

- все устройства ввода-вывода могут добавляться в компьютерную систему или извлекаться из нее при включенном компьютере;
- присоединенное к системе устройство легко конфигурируется самой операционной системой в автоматическом режиме, т. е. вмешательство пользователя здесь является минимальным;
- наличие универсального соединителя, который позволяет использовать устройство с различными системами;
- высокая скорость обмена данными по шине – 12 МБ/с все-таки намного выше, чем наивысшие скорости параллельного и последовательного интерфейсов;
- возможность подключения до 127 устройств одновременно. Практических ограничений для возможностей расширения систем на базе USB-шины не существует;
- возможность питания подсоединенных устройств непосредственно с кабельного разъема, что исключает необходимость наличия дополнительного источника питания;
- управление режимом энергопотребления: устройство USB автоматически отключается, если к нему нет обращения;
- система сразу же сообщает о наличии ошибок в обмене данными или в конфигурировании устройства. Кроме того, имеется возможность повтора неудавшейся транзакции, например, при записи данных на USB флэшдиск;

- удобство подсоединения устройства к компьютерной системе и отсоединения от нее, поскольку все разъемы USB находятся снаружи корпуса;
- интерфейс является простым с точки зрения аппаратно-архитектурных решений, что обеспечивает создание новых классов периферийных устройств для компьютерных систем.

Спецификация стандарта USB обеспечивает выбор атрибутов, при которых достигается наилучшее соотношение цена/производительность и обеспечивается выполнение функций дифференцирования на уровне компонентов и системы.

Интерфейс USB имеет ассиметричную логику и включает хост-контроллер (концентратор, хаб) и несколько устройств, соединенных в дейзи-цепочку. В эту цепочку могут быть включены дополнительные устройства USB, которые могут образовывать древовидную структуру максимум с 5-ю уровнями для одного контроллера.

Схему подключения устройства USB к компьютеру иллюстрирует рис. 6.1:

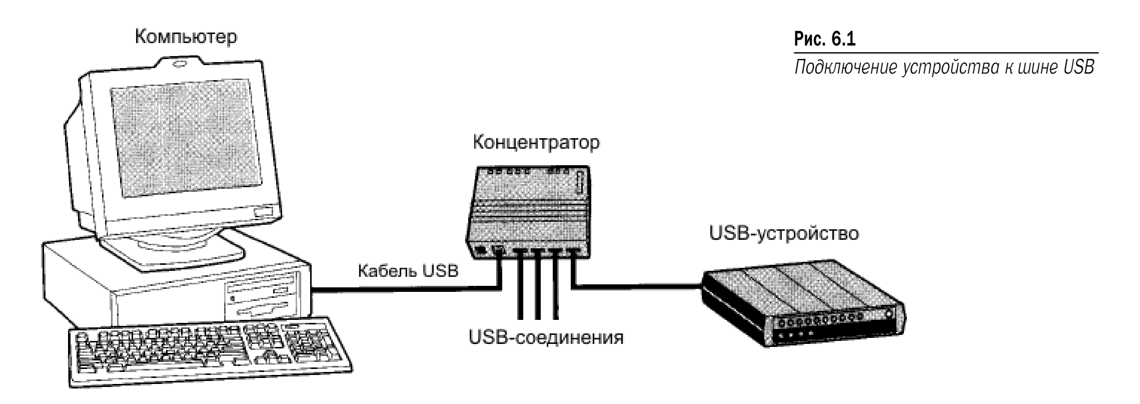


Рис. 6.1
Подключение устройства к шине USB

В типовой конфигурации устройства USB подсоединяются к компьютеру посредством USB-кабелей. На материнской плате современных компьютеров имеется интегрированный концентратор (хаб) USB, который обычно называют корневым концентратором (на рис. 6.1 для большей ясности USB-концентратор показан в виде отдельного блока).

Это очень упрощенная схема подключения USB-устройств. На самом деле концентратор управляется еще одним интегрированным в материнскую плату устройством – так называемым хост-контроллером. Хост-контроллер USB может управлять (что часто и делается) несколькими концентраторами.

В современных материнских платах, как правило, имеется несколько хост-контроллеров и несколько концентраторов, к которым подсоединяются устройства пользователя. Операционная система взаимодействует с хост-контроллером посредством портов ввода-вывода или регистров, отображаемых на память. Оконечные устройства USB (мыши, принтеры, сканеры, видеокамеры и т. д.) в терминологии USB называют «функциями».

Функциональная схема взаимодействия всех этих устройств показана на рис. 6.2.

Хост-контроллер подключается к шине PCI по стандартной схеме, как это делается для других устройств. Операционная система взаимодействует с хост-контроллером посредством портов ввода-вывода или регистров, отображаемых на память. Сам хост-контроллер управляет иерархической структурой, состоящей из концентраторов и USB-устройств, как это видно из рис. 6.2.

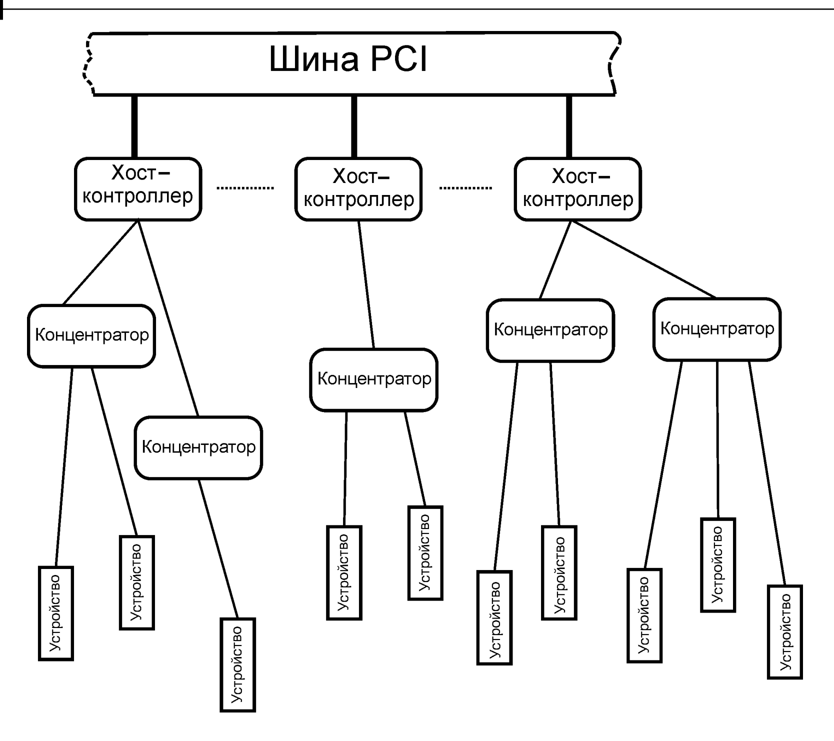


Рис. 6.2
Функциональная схема USB-шины

Как же выглядит конфигурация USB, например, в операционной системе Windows? Чтобы посмотреть это, необходимо на рабочем столе выбрать на пиктограмме **Мой компьютер** опцию **Управление**. Затем в раскрывающемся окне выбрать опцию **Диспетчер устройств**. Если в окне справа раскрыть список **Контроллеры универсальной последовательной шины USB**, то можно увидеть конфигурацию шины USB и устройства, подсоединенные к ней. Пример такой конфигурации показан на рис. 6.3.

Из этого рисунка видно, что в этой конкретной конфигурации на материнской плате имеется пять хост-контроллеров и столько же концентраторов плюс единственное устройство USB – флэш-диск. Тем не менее, здесь показана неполная конфигурация. Для того чтобы увидеть не только устройства USB, но и схему их подключения, следует в окне **Управление компьютером** выбрать опцию меню **Вид** и далее в списке установить опцию **Устройства по подключению** (рис. 6.4).

Если теперь просмотреть конфигурацию USB, то она даст нам более четкое представление о том, каким образом, к какому хост-контроллеру и концентратору подсоединены устройства USB, что может оказаться очень полезным при поиске неисправностей и настройке USB-устройств. Ниже представлен пример иерархии USB-устройств (рис. 6.5).

Как видно из этого рисунка, в системе имеется два USB-устройства: мышь и флэш-диск, которые подключены к корневому концентратору хост-контроллера USB-шины Intel 82801EB–24D2.

Наличие нескольких физических устройств в хост-контроллере позволяет выполнять параллельную обработку данных от разных USB-устройств, поскольку каждое из таких устройств имеет выделенный ему диапазон памяти и линию прерывания, что, в свою очередь, позволяет драйверу устройства выполнять обработку данных асинхронно в процедурах прерывания.

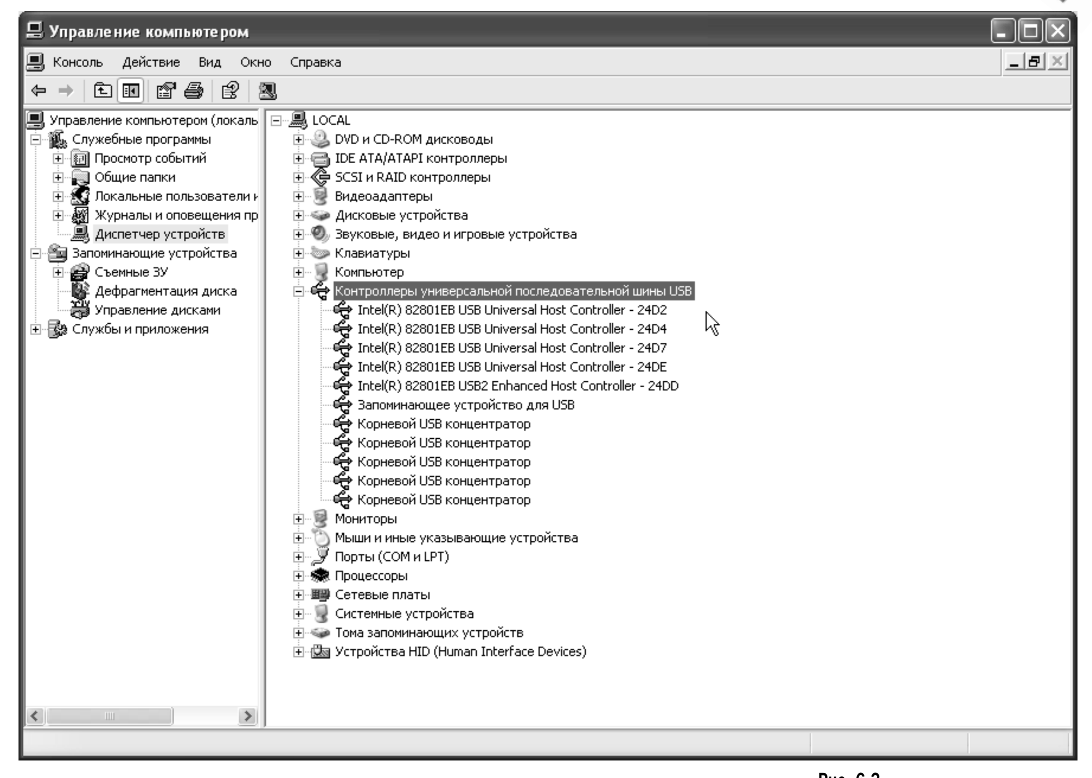


Рис. 6.3
Пример конфигурации USB-шины

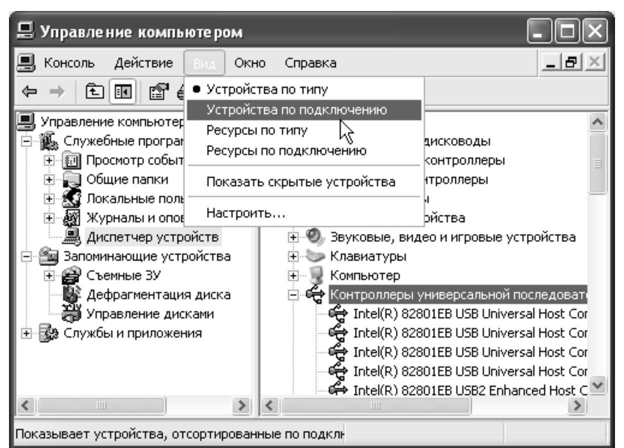


Рис. 6.4
Выбор опции для просмотра конфигурации USB-устройств

Расширить количество подключаемых к хост-контроллеру USB-устройств можно, используя каскадирование концентраторов, как показано на рис. 6.6.

Все устройства USB управляются программно либо операционной системой, либо другими программами. Для того чтобы программировать устройства, необходимо четко представлять себе аппаратно-программную модель взаимодействия устройств USB и системы. В упрощенном виде такое взаимодействие можно представить схемой, показанной на рис. 6.7.

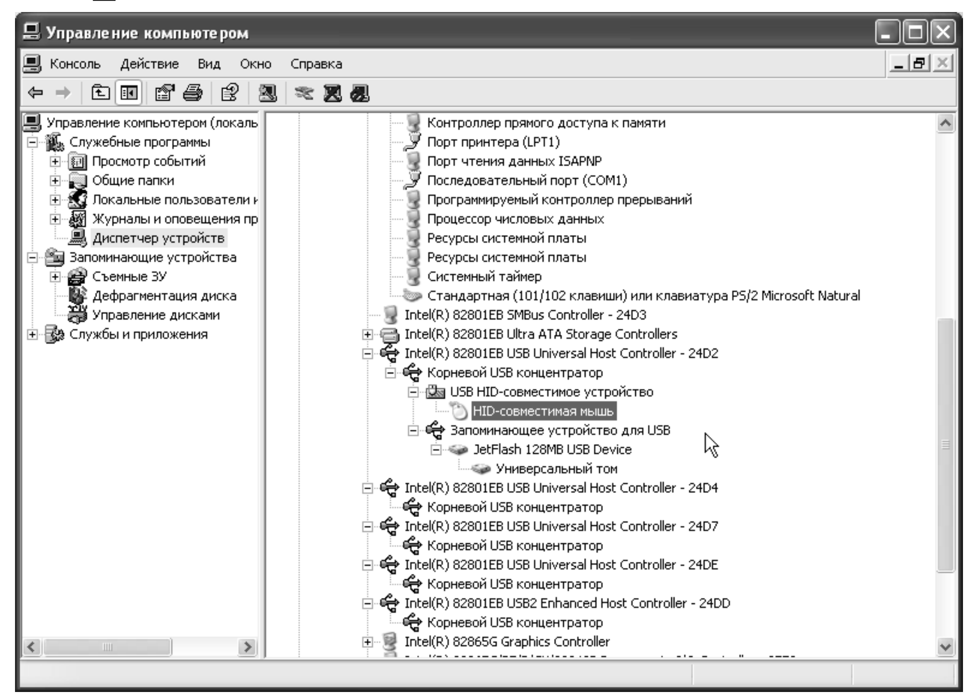


Рис. 6.5
Иерархия шины USB

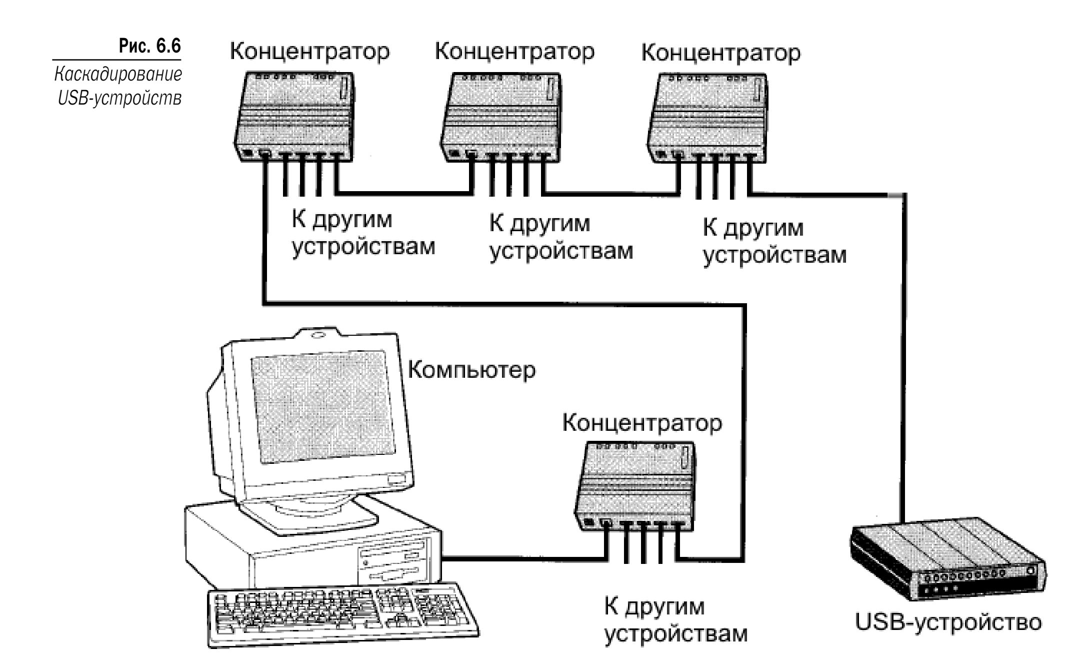


Рис. 6.6
Каскадирование USB-устройств

Как видно из схемы на рис. 6.7, взаимодействие устройства USB и компьютера можно представить в виде иерархической модели, на самом нижнем уровне которой располагается физический уровень (т. е. физическая среда передачи информации, например, кабельные

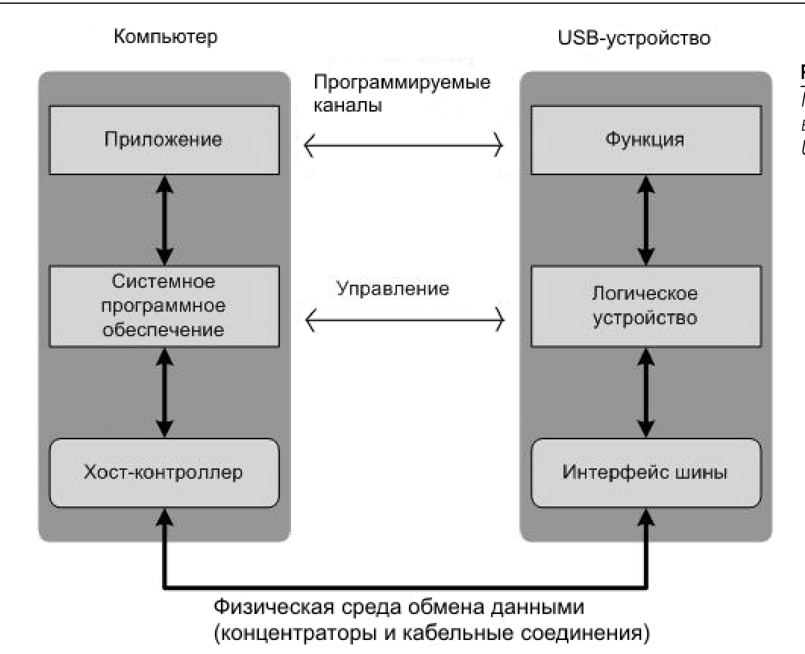


Рис. 6.7
Программная модель взаимодействия USB-устройства и системы

соединения и концентраторы). Все остальные уровни, начиная с хост-контроллера, реализованы в самой компьютерной системе, причем взаимодействие с USB-устройством может осуществляться на различных уровнях этой модели, не обязательно на самом верхнем. Подобная функциональная схема описывает программную архитектуру USB-шины для любой операционной системы. В конкретных операционных системах (Windows, Linux и т. д.) реализации такой программной архитектуры, естественно, будут несколько отличаться от общей модели.

Далее мы будем рассматривать функционирование USB-устройств применительно к операционным системам, совместимых с Windows 2000/XP/2003/Vista.

6.1. Функционирование USB–устройств в операционных системах Windows

Операционная система по отношению к USB-устройствам выполняет две основные функции: инициализацию устройств и программную поддержку их функционирования. Процесс инициализации USB-устройств может выполняться не только в момент включения персонального компьютера и первоначальной загрузки, но также и на работающей операционной системе. Это позволяет добавлять новые устройства в систему, равно как и удалять их из системы в любое время. Как только новое устройство добавляется в систему, менеджер PnP запускает программное обеспечение, отвечающее за инициализацию устройств. При этом вновь обнаруженному устройству присваивается уникальное значение идентификатора, по которому это устройство будет доступно операционной системе.

В операционных системах Windows управление USB-устройствами осуществляется посредством устройства, известного под названием «хост-контроллер». Немного раньше мы вскользь останавливались на этом устройстве, сейчас же рассмотрим его более подробно.



Хост-контроллер USB – это аппаратно-программный мост между вашими устройствами USB и операционной системой. Все запросы к устройству или все запросы от устройства проходят через хост-контроллер. Если проанализировать аппаратно-программную архитектуру USB-шины для операционных систем Windows, то ее можно детализировать так, как показано на рис. 6.8.

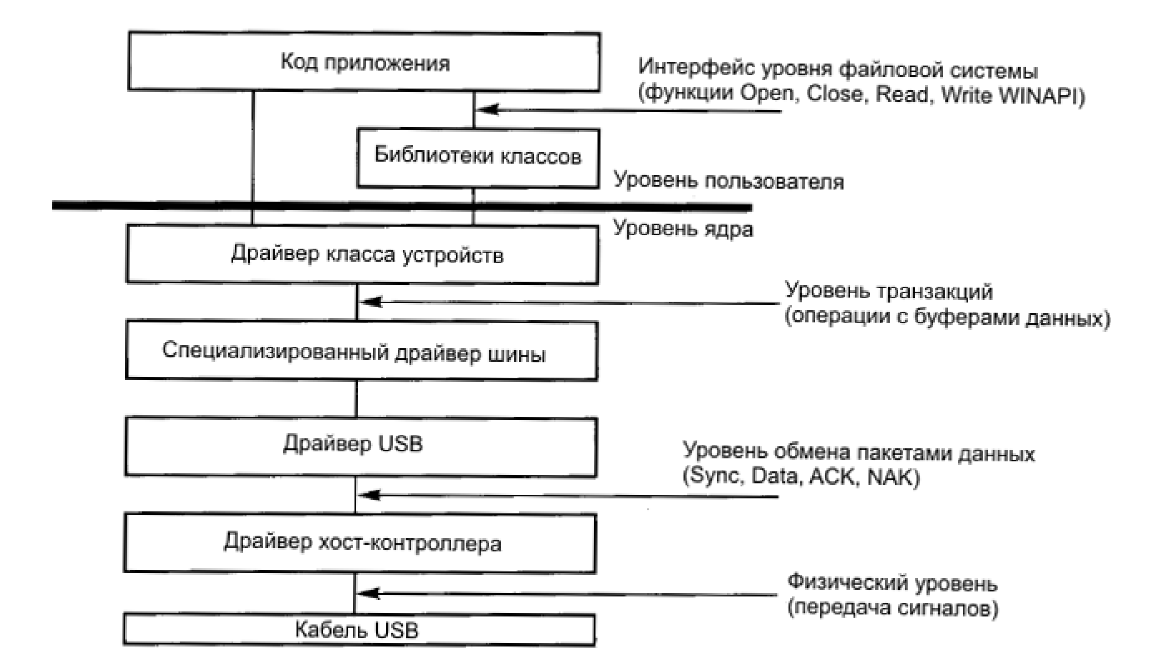


Рис. 6.8
Аппаратно-программная модель USB-шины в Windows

На рис. 6.8 показана многоуровневая структура программного обеспечения шины USB. В этой схеме class (класс) представляет собой группу устройств, обладающих общими характеристиками. Такие устройства могут контролироваться драйвером класса устройств. Примерами классов устройств являются запоминающие устройства, коммуникационные и аудиоустройства, а также устройства пользователя. Возможна ситуация, когда одно и то же устройство может принадлежать сразу к нескольким классам. Если в вашей системе имеется USB-устройство, относящееся к какому-либо классу, то разрабатывать для него драйвер нет необходимости. В этом случае может возникнуть только необходимость использовать какое-либо специальное свойство такого устройства, что, возможно, потребует написания драйвера фильтра, который будет располагаться в стеке драйверов USB над драйверами нижнего уровня.

На самом нижнем уровне этой модели взаимодействие осуществляется между хост-контроллером, подключенным к шине PCI, и шинным интерфейсом физического устройства USB. На втором уровне взаимодействие осуществляется между системным программным обеспечением и логическим устройством, представляющим более высокий уровень абстракции. Наконец, на третьем уровне осуществляется обмен данными между клиентским программным обеспечением и функцией, которая представляется данным устройством.

Фактически весь информационный поток проходит через физический канал на самом нижнем уровне, поэтому представление информационных потоков на двух верхних

уровнях является виртуальным и выделено для того, чтобы лучше понять специфику информационного обмена.

В операционных системах Windows нижний уровень информационной модели представлен группой драйверов устройств. В эту группу входит драйвер хост-контроллера (обычно это usbuhci.sys), драйвер корневого концентратора (usbhub.sys) и библиотека динамической компоновки usbui.dll, используемая системой и клиентскими драйверами (usbd.sys). Все вместе эти драйверы управляют аппаратными соединениями, создавая возможности для реализации программных каналов для взаимодействия на более высоких уровнях.

Стандарт USB определяет четыре типа передачи данных:

- Control — передача и прием управляющих сигналов (используется для конфигурирования вновь присоединенных устройств). Этот тип гарантирует обмен данными без потерь, причем размер данных может быть меньше или равен 8, 16, 32 или 64 байта;
- Bulk — передача и прием небольших пакетов неструктурированных данных. Этот тип гарантирует обмен данными без потерь, причем размер данных может быть меньше или равен 8, 16, 32 или 64 байта. Обмен данными этого типа чаще всего осуществляется при работе с принтерами или сканерами;
- Interrupt — передача данных, содержащих информацию (например, определенные символы), позволяющую получить отклик с заранее известными характеристиками. Этот тип гарантирует обмен данными без потерь, причем размер данных может быть меньше или равен 64 байтам;
- Isochronous — передача или прием больших объемов неструктурированных блоков данных с определенной, заранее установленной периодичностью. Этот тип не гарантирует отсутствие потерь при передаче данных, размер данных может быть меньше или равен 1023 байта. Типичным примером данных этого типа является голосовая информация.

Каждый канал передачи данных работает только с одним из указанных типов передаваемых данных.

Когда пользовательское приложение отправляет или получает данные через программный канал USB, то вызывается соответствующая функция Win API, которая посредством Менеджера ввода-вывода операционной системы обращается к драйверу устройства. Менеджер ввода-вывода формирует пакет запроса на ввод-вывод (I/O Request Packet, IRP) и передает его драйверу. Основной функцией драйвера является передача данных через программный канал устройству USB, при этом данные одного из типов, рассмотренных нами ранее, передаются как совокупность транзакций.

Информация в виде микрокадров (microframes) каждые 125 микросекунд передается по шине USB 2.0 (или каждую миллисекунду для USB 1.1). Каждая транзакция, в свою очередь, разбивается на фазы (одну или больше). Каждая фаза может быть представлена одним из пакетов:

- маркерный пакет или просто маркер (token) — передается хост-контроллером всем сконфигурированным на шине устройствам. Маркер включает в себя адрес устройства и, во многих случаях, номер узла. Только устройство, распознавшее адрес как собственный, будет продолжать обмен данными;
- пакет данных (data) — передается или от хост-контроллера устройству (запись данных) или принимается от устройства (чтение данных);





- квитирование (handshake) — пакет, в который записывается статусная информация (информация о состоянии обмена) — помещается на шину USB или хост-контроллером, или устройством. Например, в случае успешного приема информации устройство помещает на шину пакет типа ACK. Если устройство занято, помещается пакет NAK. Если данные успешно приняты, но по каким-то причинам нарушена логика обмена, то устройство устанавливает пакет STALL.

Схему одной транзакции при обмене данными можно представить так, как показано на рис. 6.9.

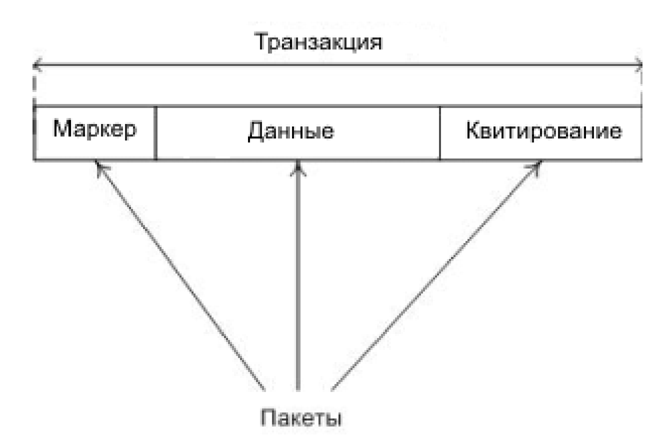


Рис. 6.9
Схема одной транзакции при обмене данных

Если внимательно проанализировать пакеты квитирования, то можно обнаружить, что в них нет детального описания характера ошибки на шине. Хост-контроллер или устройство USB не выполняют детальный анализ ошибок — если таковая имеется, то это означает, что транзакцию следует выполнить повторно.

Более детальную информацию по стандарту USB можно найти в многочисленных источниках в Интернете и в описании самого стандарта, а сейчас мы рассмотрим особенности программирования устройств USB.

6.2. Программирование USB-устройств

Как уже упоминалось, операционные системы Windows обеспечивают программную поддержку функционирования устройств, подключенных к шине USB. Обработку потоков данных устройств USB на уровне операционной системы выполняет стек стандартных драйверов, которые выполняют основные функции по управлению всеми устройствами USB и обмену данными между ними и системой.

Если вам необходимо написать программное обеспечение для какого-либо устройства USB, которое расширяло бы его возможности по обработке данных, то можно избрать один из трех возможных путей:

- написать собственный драйвер устройства, который бы обеспечивал все необходимые функции управления и обмена данными, и программу, которая бы взаимодействовала с этим драйвером в пользовательском режиме. При этом можно полностью обойтись без стандартных драйверов системы;

- написать драйвер фильтра, который бы обеспечивал требуемую функциональность, но располагался бы в стеке драйверов над системными драйверами. Таким образом, все стандартные функции обработки выполняли бы драйверы USB, установленные системой, а дополнительные функции обеспечивались бы вашим драйвером фильтра, с которым и взаимодействовала бы программа пользователя;
- воспользоваться свободно распространяемыми библиотеками функций и драйверами для доступа к USB-устройству.

В большинстве случаев программный доступ к устройству USB может потребоваться, если данное устройство выполняет какую-то очень специфичную функцию. Например, на базе USB разработаны «электронные осциллографы» или системы сбора данных, для работы с которыми необходимо иметь доступ к самому устройству. В большинстве таких случаев можно воспользоваться свободно распространяемыми библиотеками функций, которые будут работать практически во всех популярных средах программирования. Например, под эгидой GNU разработано программное обеспечение, известное под названием LibUsb, включающее необходимые драйверы и библиотеки функций для работы в операционных системах Windows и Linux. Эти библиотеки функций очень популярны и позволяют быстро разрабатывать программы, взаимодействующие с вашим устройством посредством набора стандартных функций. Это исключает необходимость написания собственного драйвера устройства, что существенно экономит время.

Кроме того, большинство пользователей не знакомо с методикой разработки драйверов, а это очень сложная область программирования, поэтому наличие такого свободно распространяемого программного обеспечения окажет неоценимую помощь широкому кругу пользователей. На основе проекта LibUsb разработаны оболочки (wrappers) для работы с Visual Basic .NET и C# .NET, наиболее популярной из которых является LibUsbDotNet, также разработанная под эгидой свободно распространяемого программного обеспечения. Несмотря на кажущуюся сложность программирования USB-устройств, перечисленное программное обеспечение настолько упрощает эту задачу, что она становится под силу даже новичкам. Рассмотрим на практических примерах, как работать с вашими USB-устройствами, и начнем с пакета программ LibUsb. Кстати, вышеперечисленное программное обеспечение можно бесплатно загрузить с сайта www.sourceforge.net или из многочисленных дублирующих сайтов.

Как работать с библиотеками USB-функций LibUsb? Библиотека построена таким образом, чтобы можно было выполнять основные операции, связанные с USB-устройством:

- идентификацию или, по–другому, перечисление (enumeration). При выполнении этой операции происходит обнаружение устройств, подключенных к шине USB, что выполняется с помощью соответствующих функций библиотеки libusb;
- получение параметров устройства (идентификаторов устройства, данных о производителе и характеристиках устройства), для чего в библиотеке имеется целый ряд функций;
- открытие, закрытие, чтение и запись данных, посылка команд. К устройству USB, так же, как и к другим объектам файловой системы, можно обращаться по записи-чтению, что и выполняется с помощью соответствующих функций библиотеки.

Все перечисленные возможности могут быть реализованы посредством вызова соответствующих функций библиотеки libusb, но здесь они не будут перечисляться, поскольку это заняло бы слишком много места; мы посмотрим, как использовать некоторые из этих функ-



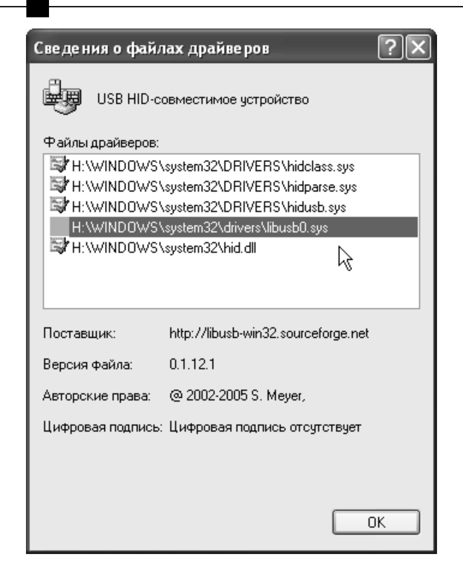


Рис. 6.10
Расположение драйвера libusb0.sys в стеке драйверов устройства

ций на практических примерах. Описание всех функций читатели смогут найти в соответствующей документации. Напомню, что мы рассматриваем применение функций библиотеки libusb в операционных системах Windows.

При инсталляции дистрибутива с libusb в операционной системе Windows в системе инсталлируется драйвер фильтра libusb0.sys. Этот драйвер будет находиться в вершине стека драйверов системы, что легко увидеть, например, посмотрев сведения о драйверах для любого USB-устройства (рис. 6.10).

Кроме того, для обращения к драйверу из программ пользователя в систему инсталлируется библиотека libusb0.dll, используя которую можно разрабатывать пользовательские программы.

Посмотрим, например, как получить информацию о количестве активных устройств USB в системе и количестве интерфейсов, к которым подключены устройства. Для этого разработаем в Visual Basic .NET простое приложение, в котором будут использоваться функции библиотеки libusb (libusb к этому моменту должна быть установлена в системе).

Для тестируемой системы количество подключенных USB-устройств равно 3 (рис. 6.11).

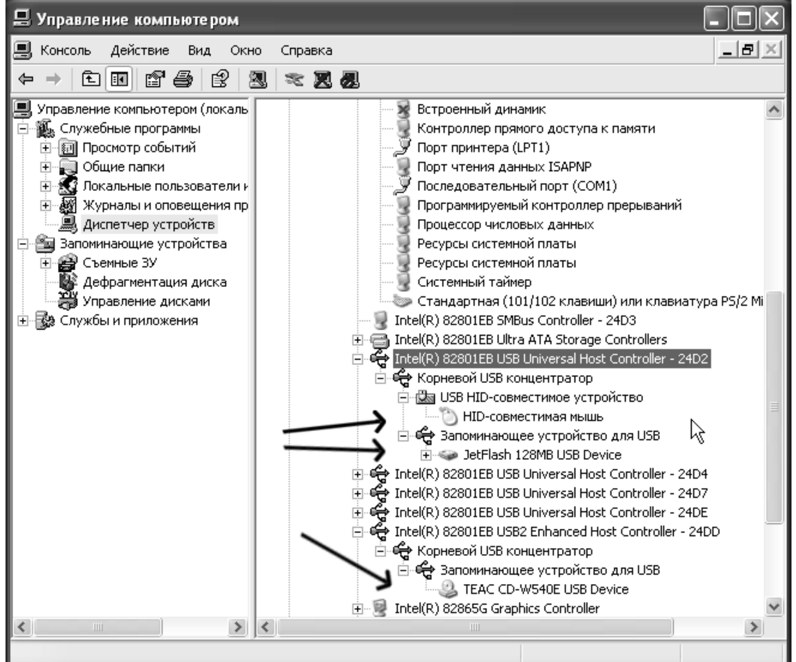


Рис. 6.11
Пример конфигурации USB-устройств



Разработаем простое графическое приложение в Visual Basic .NET, окно конструктора которого будет выглядеть следующим образом (рис. 6.12):

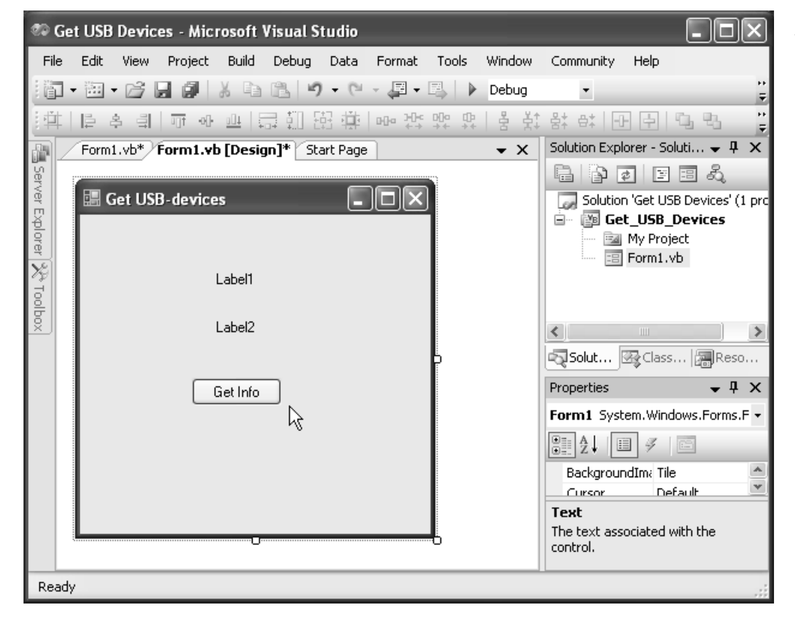


Рис. 6.12
Окно конструктора приложения

Исходный текст приложения очень прост и содержит всего несколько строк:

```
Public Class Form1
  Declare Sub usb_init Lib "libusb0.dll" ()
    Declare Function usb_find_busses Lib "libusb0.dll" () As Integer
    Declare Function usb_find_devices Lib "libusb0.dll" () As Integer
   Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
   Dim nBusses As Integer
   Dim nDevices As Integer

   usb_init()
   nBusses = usb_find_busses()
   nDevices = usb_find_devices()
   Label1.Text = "Devices " & nDevices
   Label2.Text = "Busses " & nBusses

  End Sub
End Class
```

Здесь вначале программы с помощью директив Declare объявляются функции usb_init, usb_find_busses и usb_find_devices из библиотеки динамической компоновки libusb0.dll. Функция usb_init выполняет инициализацию устройств шины, функция usb_find_busses воз-



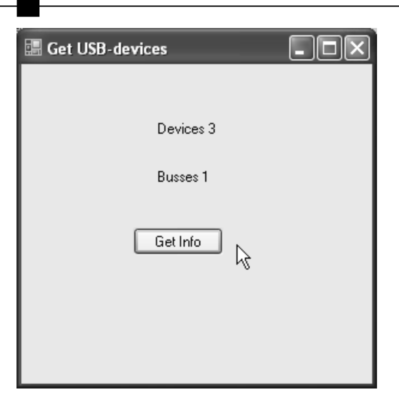


Рис. 6.13
Вид окна работающей программы

вращает количество задействованных USB-шин, а функция usb_find_devices возвращает количество активных USB-устройств.

При нажатии на кнопку в текстовых метках label1 и label2 будет отображена соответствующая информация. В нашем примере при запуске программа выдаст результат, показанный на рис. 6.13.

Для читателей, работающих в C# .NET, исходный текст приложения будет таким:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Runtime.InteropServices;

namespace CSharp_USB
{
  public partial class Form1 : Form
  {
   public int nDevices;
   public int nBusses;

   [DllImport("libusb0.dll")]
     public static extern void usb_init();
   [DllImport("libusb0.dll")]
     public static extern int usb_find_busses();
   [DllImport("libusb0.dll")]
     public static extern int usb_find_devices();

   public Form1()
    {
     InitializeComponent();
    }

   private void button1_Click(object sender, EventArgs e)
    {
     usb_init();
     nBusses = usb_find_busses();
     nDevices = usb_find_devices();
     label1.Text = "Busses: " + nBusses;
     label2.Text = "Devices: " + nDevices;
   }
  }
}
```

В среде программирования C# .NET для работы с функциями из библиотеки динамической компоновки следует импортировать пространство имен System.Runtime.InteropServices, после чего необходимо объявить функции библиотеки директивой DllImport.

Более сложный пример, написанный в Visual C++ .NET, позволяет получить информацию обо всех USB-устройствах. Вот исходный текст консольного приложения:

```
#include <stdio.h>
#include <windows.h>
#include <usb.h>

#define DEBUG_LEVEL 0
#define PROTOCOL_MOUSE 2
#define PROTOCOL_KEYBOARD 1

struct usb_bus *busses,*bus,*dbus;

int findDevice(int PROTOCOL,struct usb_device *device)
{
  bool found = false;
  struct usb_device *dev;

  usb_find_busses();
  usb_find_devices();
  busses = usb_get_busses();

  for (bus = busses;bus && !found;bus=bus->next)
   {
    for (dev=bus->devices;dev; dev = dev->next)
    {
    if (dev->config->interface->altsetting->bInterfaceProtocol==PROTOCOL)
     {
      dbus=bus;
      found=true;
       break;
     }
   }
 }
if (!found)
   return -1;
device = dev;
return 1;
}


void main(void)
{

 struct usb_device *dev;
 char *buf;
```

```
 int n,x,r;
 char string[50];
 bool found = false;

 usb_dev_handle *fdev;
 usb_set_debug(DEBUG_LEVEL);
 usb_init();

 if (findDevice(PROTOCOL_MOUSE,dev)<0)
  {
   printf("Unable to find the required device !");
   exit(1);
  }
 printf("Trying to open the device...n");

 if (fdev=usb_open(dev))
  printf("Device opened successfully.n");
 else
  {
   printf("Operation failed :-(n");
   exit(1);
  }
 buf=(char*)calloc(1,100);

 if (r = usb_claim_interface(fdev,0))
      printf("Interface Claimed !!n");

 printf("Interface Claim Status : %dn",r);
 printf("Device Protocol : %dn",dev->descriptor.bDeviceProtocol);

 printf("Report Length : %dn",dev->descriptor.bLength);
 printf("Decriptor Type : %dn",dev->descriptor.bDescriptorType);

 printf("End Points :%dn",dev->config->interface->altsetting-
>bNumEndpoints);
 printf("Interface Class :%dn",dev->config->interface->altsetting-
>bInterfaceClass);

 printf("Protocol :%dn",dev->config->interface->altsetting-
>bInterfaceProtocol);
 printf("Interface Number:%dn",dev->config->interface->altsetting-
>bInterfaceNumber);

 printf("Device Filename : %sn",dev->filename);
 printf("Bus Dir Name : %sn",dbus->dirname);

 usb_get_string_simple(fdev,dev-
>descriptor.iManufacturer,string,sizeof(string));

 printf("Device Manfucaturer : %sn",string);
```

```
 usb_get_string_simple(fdev,dev-
>descriptor.iProduct,string,sizeof(string));

 printf("Product Name : %sn",string);

 usb_get_string_simple(fdev,dev-
>descriptor.iSerialNumber,string,sizeof(string));

 printf("Device Serial Number: %sn",string);
 printf("End point addresses : 0x%xn",dev->config->interface->altsetting-
>endpoint->bEndpointAddress);

 printf("Closing Device.n");

 usb_release_interface(fdev,0);
 usb_close(fdev);
 return;
}
```

Для того чтобы функции libusb были доступны, в проект следует добавить файл libusb.lib.

Как видно из приведенных примеров, библиотека libusb намного упрощает программирование USB-устройств. Тем не менее, в приложениях на Visual Basic и C# .NET еще более удобно было бы использовать пространство имен вместо библиотек функций. Для такого применения и был разработан GNU-проект LibUsbDotNet (www.sourceforge.net), в котором возможности библиотек libusb были перенесены на платформу NET. Теперь для программирования USB-устройства можно использовать пространство имен LibUsbDotNet.

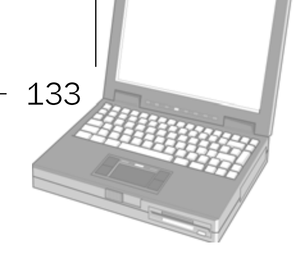
Лучше всего возможности LibUsbDotNet показать на примере. В следующем приложении, разработанном на Visual C# .NET, фиксируется добавление/удаление устройства USB с выводом информации об устройстве в окно приложения. Разместите на главной форме приложения экземпляр компонента TextBox из палитры компонентов Toolbox и задайте свойство Dock этого компонента равным Fill, тогда текстовое окно будет полностью привязано к форме. Далее показано, как выглядит форма приложения в конструкторе (рис. 6.14).

Для работы с компонентом LibUsbDotNet следует указать ссылку на файл, содержащий данный компонент (для NET-компонентов это, как правило, dll-файл), для чего в окне **Add Reference** выбрать опцию меню **Browse** и найти файл компонента LibUsbDotNet (рис. 6.15).

Затем необходимо импортировать это пространство имен в наш проект, добавив его с помощью директивы using. Полный исходный текст программы показан далее:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Windows.Forms;
using LibUsbDotNet.DeviceNotify;
using LibUsbDotNet.Usb;
using LibUsbDotNet.Usb.Main;

namespace Test_DeviceNotify
{
```



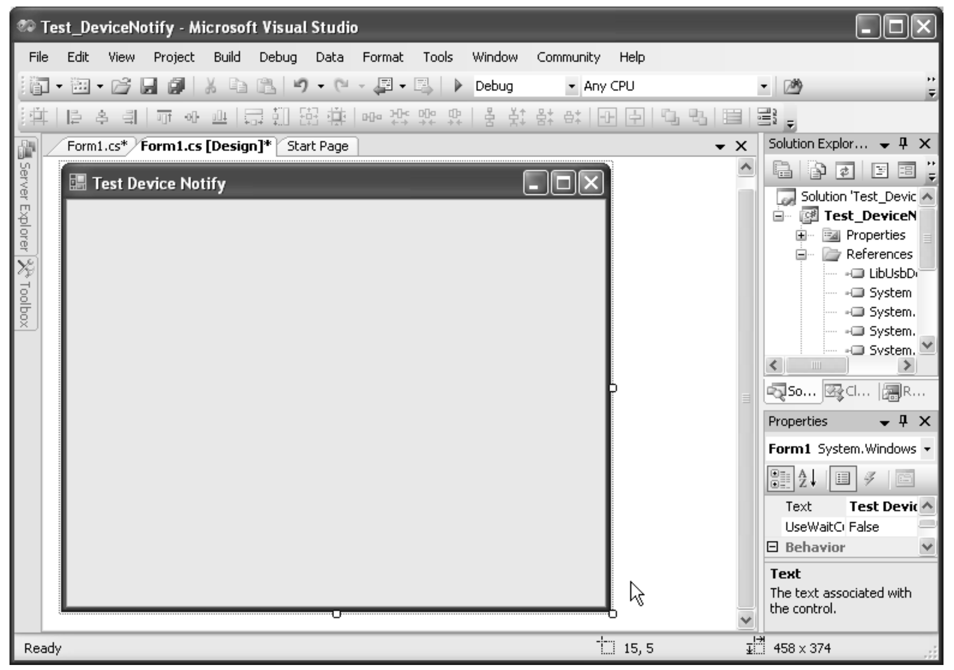


Рис. 6.14
Окно конструктора приложения

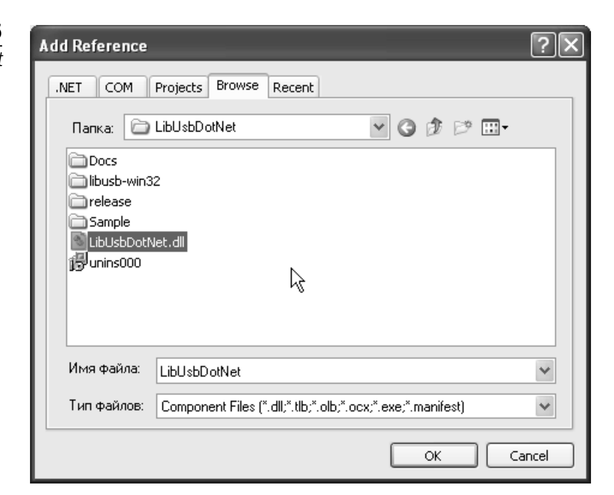


Рис. 6.15
Выбор файла компонента LibUsbDotNet

```
public partial class Form1 : Form
  {
   private DeviceNotifier mDevNotifier;

   public Form1()
    {
     InitializeComponent();
     mDevNotifier = new DeviceNotifier();
```

```
       mDevNotifier.OnDeviceNotify += new
EventHandler<DeviceNotifyEventArgs>(mDevNotifier_OnDeviceNotify);
      }

  #region PRIVATE METHODS

 private void mDevNotifier_OnDeviceNotify(object sender,
DeviceNotifyEventArgs e)
   {
     Invoke(new DeviceNotifyEventDelegate(OnDevNotify), new object[]
{sender, e});
     Debug.Print(e.Object.ToString());
   }

   private void OnDevNotify(object sender, DeviceNotifyEventArgs e)
    {
     object[] o = new object[] {e.EventType.ToString(),
DateTime.Now.ToString(), e.DeviceType.ToString(), e.Object.ToString()};
     string s = String.Format("{0} - Time: {1}  -  {2}\r\n{3}", o);

     if (e.DeviceType == DeviceType.DEVICEINTERFACE && e.EventType ==
EventType.DEVICEARRIVAL)
             s += "\r\n" + e.Device.SymbolicName.FullName;
     tNotify.Text += s;
    }

  #endregion

  #region Nested type: DeviceNotifyEventDelegate

  private delegate void DeviceNotifyEventDelegate(object sender,
DeviceNotifyEventArgs e);

  #endregion
 }
}
```

В этой программе при добавлении/удалении USB-устройства вызывается обработчик события DeviceNotify, который будет выводить в текстовое окно информацию о добавляемом/удаляемом устройстве.

Если запустить наше приложение и подключить USB-устройство, например, CD–RW, то в окне приложения появится соответствующее сообщение (рис. 6.16).

Если теперь удалить USB-устройство из системы, то в окне приложения появится другое сообщение (рис. 6.17).

Свойства и методы компонента LibUsbDotNet хорошо документированы и доступны после инсталляции компонента в систему. Читатели, интересующиеся применением компонента, могут найти источники информации и в Интернете.

Следует отметить, что многие фирмы выпускают программное обеспечение, подобное вышеописанному, но далеко не бесплатное. Тем не менее, возможности libusb ничуть не уступают коммерческим программам.



Рис. 6.16

Вид окна приложения при добавлении USB-устройства в систему

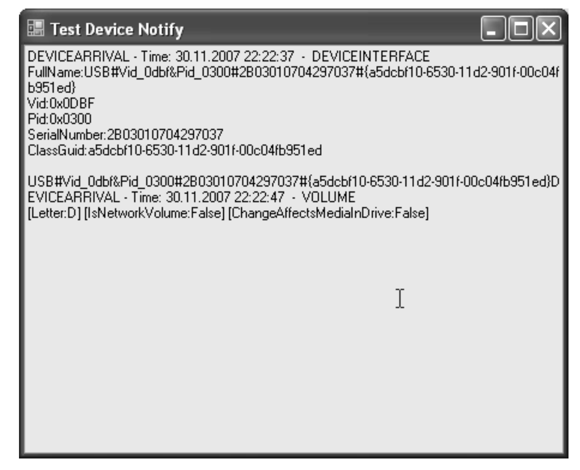


Рис. 6.17

Вид окна приложения при удалении USB-устройства из системы



6.3. Устройства Bluetooth и их программирование

Технология Bluetooth является одной из бурно развивающихся отраслей беспроводных компьютерных технологий, получивших очень широкое распространение в последнее время. Вряд ли среди читателей этой книги найдется тот, кто хотя бы не слышал о Bluetooth. Эта технология широко применяется в мобильных телефонах, карманных компьютерах, а также во многих других устройствах. Эта технология расширяет возможности и такой популярной шины, как USB (на рынке присутствует много устройств, реализованных как dongle (заглушка), который вставляется в стандартный разъем USB).

Мы рассмотрим как архитектуру Bluetooth, так и средства программирования таких устройств, наиболее распространенным из которых является язык Java, для которого разработан программный интерфейс Bluetooth JSR–82.

Что же представляет собой технология Bluetooth? В двух словах, Bluetooth – коммуникационный протокол беспроводной связи, обеспечивающий взаимодействие одних устройств



Bluetooth с другими. В этом смысле этот протокол напоминает другие коммуникационные протоколы (HTTP, FTP, SMTP и т. д.), которые используются для обмена данными и сообщениями. Протокол Bluetooth предназначен для работы в архитектуре «клиент-сервер»: устройство, инициирующее соединение, является клиентом, а устройство, ожидающее соединения, называется сервером. Bluetooth представляет собой очень эффективный протокол для беспроводных коммуникаций, поскольку при скорости передачи данных около 1 МБ/с он требует мощности в 100 раз меньше, чем, например, Wi-Fi.

На сегодняшний день Bluetooth используется в различных цифровых устройствах, таких как сотовые телефоны, цифровые видеокамеры, персональные органайзеры, карманные компьютеры и т. д.

Рабочий радиус действия Bluetooth-соединений ограничивается 100 метрами, но во многом зависит от условий эксплуатации. Интерфейс Bluetooth позволяет передавать голосовые сообщения (64 Кбит/с), а также данные, при этом передача данных может быть как асимметричной (721 Кбит/с в одном направлении и 57,6 Кбит/с в другом), так и симметричной (432,6 Кбит/с в обоих направлениях). Приемо-передающее устройство Bluetooth работает на частоте 2.4 ГГц и позволяет в зависимости от уровня мощности устанавливать связь в пределах 10-100 метров. Чем меньше расстояние, тем меньше потребляемая мощность, при этом для расстояний в пределах 10 м мощность будет минимальной при компактном размере и невысокой цене самого устройства. Так, микромощный передатчик в режиме ожидания потребляет ток всего 0.3 мА и около 30 мА при передаче информации.

Протокол Bluetooth использует разбивку данных на пакеты (алгоритм FHSS, Frequency-Hopping Spread Spectrum) и выполняет их передачу по псевдослучайному алгоритму, при котором частота меняется скачкообразно (1600 раз в секунду) или с использованием дискретного набора частот, включающего 79 субчастот. Таким образом, выполняется кодирование данных – взаимодействовать между собой могут только те устройства, которые настроены на один и тот же алгоритм передачи – другие устройства не смогут идентифицировать и расшифровать передаваемую информацию.

Как выглядят Bluetooth-устройства?

В целом, большинство USB Bluetooth-устройств очень похожи на USB флэш-диски. Вот примерный вид одного из таких устройств (рис. 6.18).

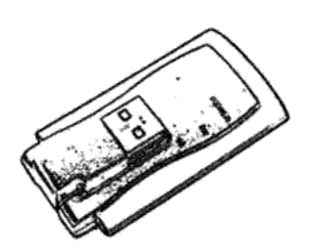


Рис. 6.18
Один из вариантов конструкции Bluetooth-устройства

Обычно инсталляция Bluetooth-устройств в компьютерной системе практически не отличается от установки любых других USB-устройств, хотя драйвер устройства потребуется устанавливать вручную, либо с компакт-диска (если есть), либо закачав его из Интернета. Если инсталляция выполнена успешно, то на нижней панели появится пиктограмма Bluetooth, посредством которой можно настраивать различные параметры устройства (рис. 6.19).

Далее можно установить дополнительные параметры Bluetooth. Выбрав первую закладку в меню конфигурации, пользователь может назначить имя компьютера, выбрать тип системы (настольный компьютер или ноутбук), а также установить уровень безопасности (рис. 6.20).

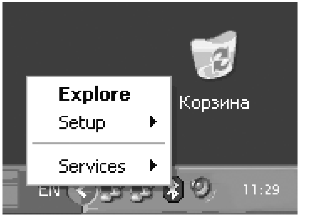


Рис. 6.19
Пиктограмма настройки Bluetooth

С помощью второй закладки можно установить возможность поиска других активных Bluetooth-устройств. Кроме того, здесь можно установить определенные правила функционирования для обнаруженных устройств (рис. 6.21).







Рис. 6.20
Окно установки параметров Bluetooth



Рис. 6.21
Установка опций для поиска других устройств

Третья закладка позволяет активизировать поиск новых Bluetooth-устройств через определенное количество минут (в данном случае, 10) (рис. 6.22).

Следующая закладка позволяет установить каталоги, где будут храниться файлы, переданные с помощью различных сервисов (рис. 6.23).

С помощью Bluetooth-устройств можно создать относительно недорогую беспроводную локальную сеть. При этом архитектура Bluetooth позволяет связать вместе несколько устройств при помощи одного шлюза. Просмотреть конфигурацию сети можно, открыв окно My Bluetooth Places. Здесь же можно изменить параметры конфигурации, посмотреть ло-

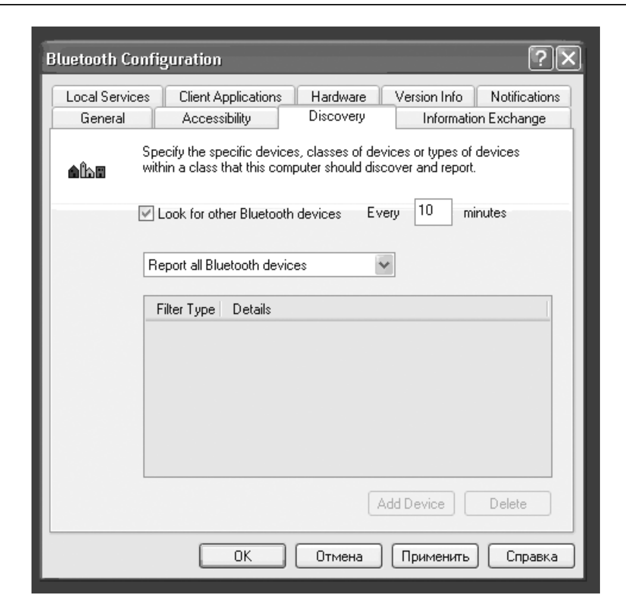


Рис. 6.22
Установка параметров поиска новых устройств Bluetooth



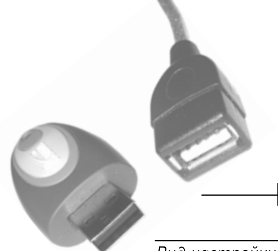
Рис. 6.23
Каталоги, в которых будут храниться файлы

кальную Bluetooth-сеть, а также включить/отключить и посмотреть статус различных служб (рис. 6.24).

Все современные Bluetooth-устройства поддерживаются стандартными службами:

- Bluetooth Serial Port – позволяет установить беспроводную Bluetooth-связь между двумя устройствами;
- Dial-Up Networking – позволяет устройству Bluetooth, выступающему в роли клиента, организовать dial–up соединение с Интернетом. Данная служба используется при





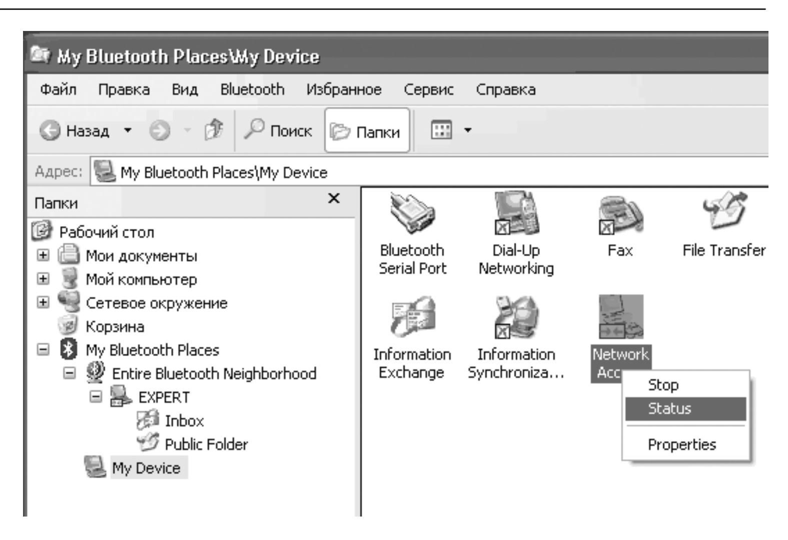


Рис. 6.24
Вид настройки окружения Bluetooth

подключении к Интернету через мобильный телефон, оснащенный Bluetooth и модемом, либо для связи через модем, подключенный к компьютеру;

- Fax – позволяет передавать факсимильные сообщения через факс-устройство, физически соединенное с сервером Bluetooth;
- File Transfer – позволяет передавать файлы между двумя Bluetooth-устройствами (рис. 6.25);

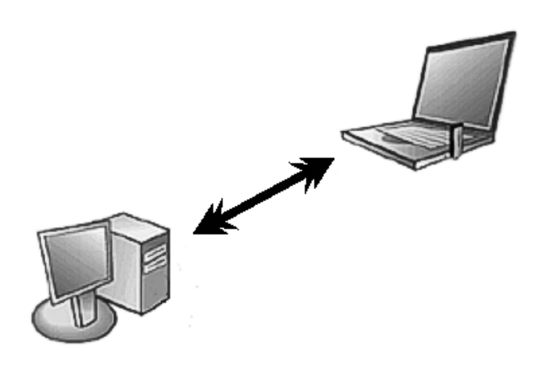
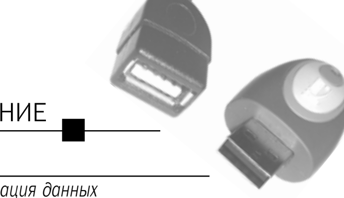


Рис. 6.25
Взаимодействие двух компьютеров посредством Bluetooth

- Information Exchange – позволяет обмениваться информацией между информационными менеджерами (органайзерами);
- Information Synchronization – позволяет организовать синхронизацию данных между персональными информационными менеджерами (рис. 6.26);
- Network Access – позволяет устанавливать беспроводную связь между клиентом и сервером, который физически связан с локальной сетью (рис. 6.27).

Чаще всего технологии Bluetooth, благодаря простому конфигурированию устройств, используют для быстрой организации беспроводного выхода в Интернет. Например, в операционной системе Windows XP последовательность шагов будет выглядеть так, как описано далее.



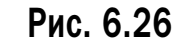


Рис. 6.26

Синхронизация данных между КПК и ПК с помощью Bluetooth

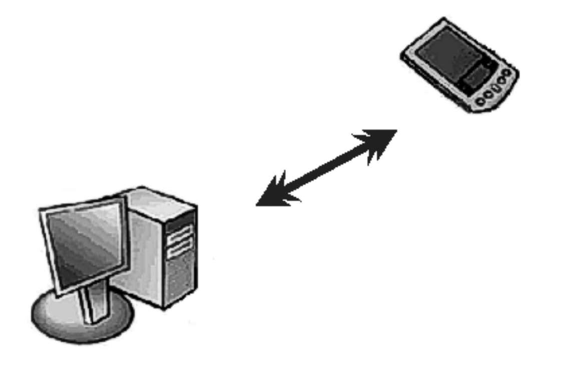
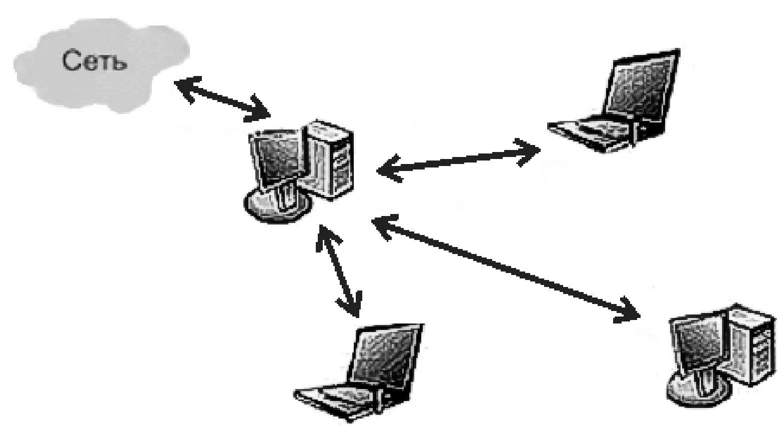


Рис. 6.27

Организация связи с внешней сетью через прокси-сервер



Создание беспроводного доступа в Интернет занимает несколько минут. На первом шаге необходимо настроить сетевое соединение Bluetooth LAN Access Server Driver на сервере, выполняющем роль прокси при подключении к Интернету (рис. 6.28).

Далее следует настроить свойства вашего сетевого соединения. Эта процедура ничем не отличается от настройки параметров сетевых соединений для сетевых карт. Пример конфигурирования Bluetooth-соединения показан на рис. 6.29 и 6.30.

Выбор IP-адреса, показанный на рис. 6.30, в каждом конкретном случае определяется параметрами вашей сети (номером сети и маской). После установки параметров TCP/IP можно активизировать общий доступ к Интернету, для чего следует войти на страницу свойств сетевого соединения, которое имеет физическое соединение с Интернетом. В данном случае Интернет подключен к сетевой карте 3Com 3C996B (рис. 6.28, Local Area Connection). Затем необходимо открыть закладку **Дополнительно** и отметить позиции, как показано на рис. 6.31.

На этом настройку серверной части можно считать законченной. Теперь необходимо настроить клиентское соединение. В отличие от серверного соединения, здесь не требуется настройка TCP/IP протокола, необходимо только, чтобы было установлено автоматическое присвоение IP-адресов. Установим скорость для BluetoothNullConnection. Здесь можно выбрать максимальное значение 921600 (~1Mb) (рис. 6.32).



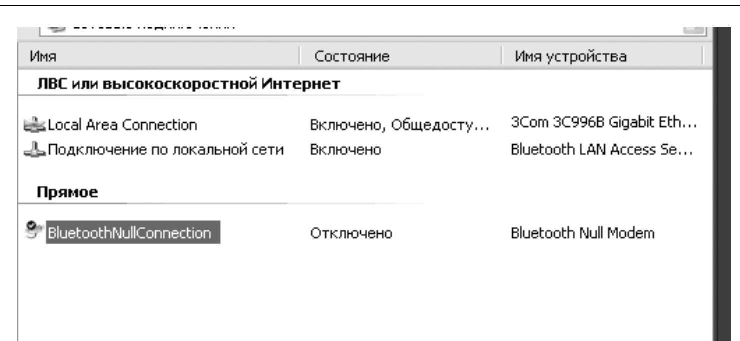


Рис. 6.28

Вид окна сетевых подключений

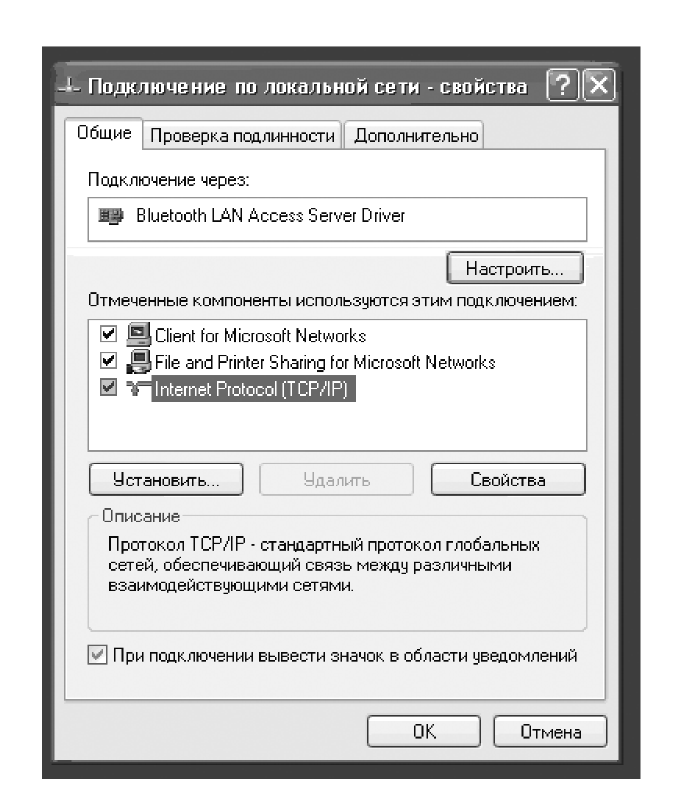


Рис. 6.29

Окно настройки сетевых параметров Bluetooth LAN Access Server Driver

На этом настройка клиентской части считается законченной, и можно попробовать выйти в Интернет, для чего необходимо сделать двойной щелчок на пиктограмме BluetoothNullConnection. Если соединение успешно установлено, то на нижней панели появятся значок установленного соединения и скорость связи.

Как видно из описания, установить и использовать Bluetooth – несложно.

6.4. Программирование Bluetooth

Для того чтобы устройства Bluetooth могли взаимодействовать друг с другом, они должны функционировать в соответствии со спецификациями протокола Bluetooth. Спецификации

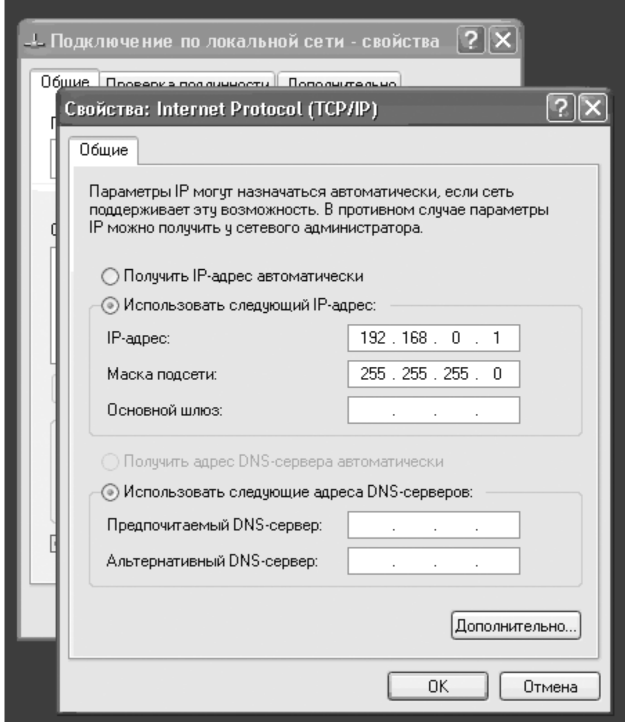


Рис. 6.30
Установка IP-адреса

Bluetooth, как и многие другие подобные протоколы, определяют стандарт, которого должно «придерживаться» устройство Bluetooth, так же как и определенных правил, которые необходимо выполнять в процессе коммуникаций. Спецификации протокола Bluetooth можно загрузить с сайта www.bluetooth.com. В спецификации протокола входят такие понятия, как «стек» и «профили». Рассмотрим более подробно смысл этих терминов.

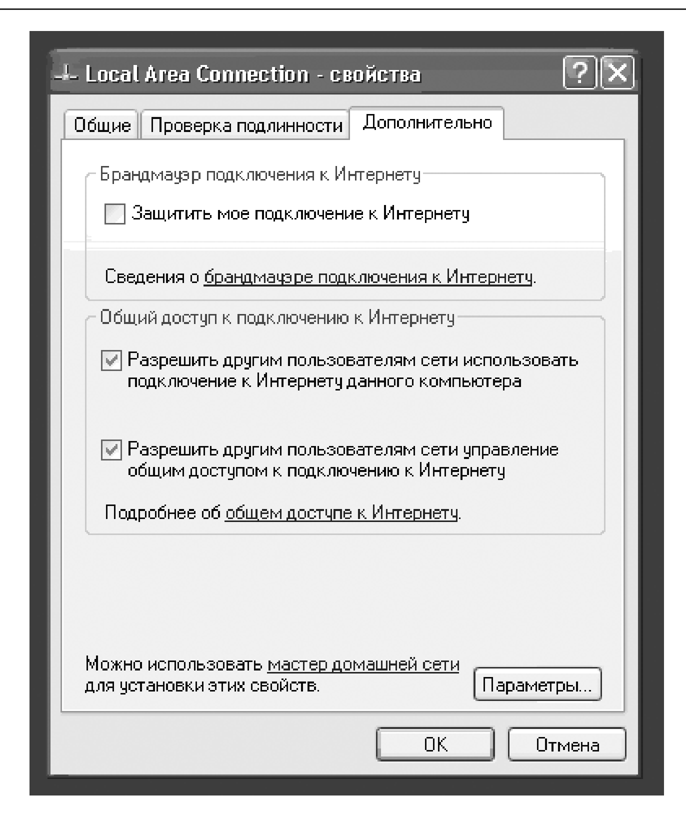
Стек протокола Bluetooth

Стек протокола представляет собой программный или аппаратно-программный компонент, позволяющий выполнить прямой доступ к Bluetooth-устройству. Он управляет различными свойствами, такими как параметры устройства, параметры коммуникации, уровни потребления мощности. Стек сам по себе состоит из уровней, каждый из которых выполняет определенную функцию из множества функций Bluetooth-устройства. Поскольку производители оборудования не обязательно используют все уровни стека в своих устройствах Bluetooth, то мы рассмотрим только основные, которые включены почти в любое устройство.

Вот основные уровни стека Bluetooth:

- HCI (Host Controller Interface) – этот уровень описывает интерфейс между радиоканалом и хостом;
- L2CAP (Logical Link Controller Adaptation Protocol) – этот уровень отвечает за мультиплексирование данных, проходящих через устройство (тем не менее, аудиосигналы напрямую проходят к HCI, минуя этот уровень);

Рис. 6.31
Установка дополнительных свойств сетевого соединения



- SDP (Service Discovery Protocol) – этот уровень используется для поиска сервисов на удаленных Bluetooth-устройствах;
- RFCOMM – больше известен как протокол «виртуального» последовательного порта.

Профили Bluetooth

Профили Bluetooth были созданы для того, чтобы различные устройства Bluetooth могли между собой взаимодействовать. Предположим, что у Вас имеется карманный компьютер с Bluetooth и мобильный Bluetooth-телефон, причем оба устройства поддерживают стек Bluetooth. Каким образом указать, можно ли синхронизировать списки телефонов в обоих устройствах? Как вы узнаете, можно ли отправить телефонный номер с вашего компьютера на мобильный телефон? И, что самое важное, как вы определите, можно ли выйти в Интернет с вашего карманного компьютера, используя телефон в качестве беспроводного модема?

Профиль Bluetooth – это встроенный набор функций устройства. Например, в только что рассмотренном примере, и мобильный телефон, и карманный компьютер должны поддерживать профиль Synchronization Profile для того, чтобы синхронизировать данные между собой. Чтобы отправить целевые данные, например, vcf-файл из карманного компьютера на телефон, оба устройства должны поддерживать профиль Object Push Profile. Наконец, компьютер и беспроводной телефон должны поддерживать профиль Dialup Networking Profile, чтобы можно было подключаться с компьютера к Интернету через телефон. Если же вы полагаете,

Рис. 6.32
Настройка клиентского Bluetooth-соединения

что для взаимодействия Bluetooth-устройств стандартные профили не подходят, то можете настроить для их взаимодействия какой-нибудь специальный профиль.

Здесь важно заметить следующее: не смешивайте профили Bluetooth с профилями программного пакета J2ME. Профили J2ME представляют собой классы Java, которые расширяют функциональность J2ME. Например, MID-профиль представляет собой множество классов Java, расширяющих функциональность CLDC (Connected Limited Device Configuration). С другой стороны, профиль Bluetooth можно реализовать на любом языке и на любой платформе, поскольку он ссылается на предопределенный набор функций Bluetooth-устройства. Например, Object Push Profile можно реализовать для операционной системы Palm PDA на языке C++, в то время как для принтера Bluetooth этот профиль можно написать на ассемблере.

Основы программирования устройств Bluetooth на языке Java

В качестве инструмента программирования Bluetooth-устройств чаще всего используется язык Java. Это обусловлено тем, что для поддержки этой технологии ведущими фирмами разработаны очень мощные и легкие в применении Java-инструментальные средства. В примерах программного кода мы будем использовать очень мощный и очень популярный программный пакет Atinav Java Bluetooth SDK (можно использовать и другие программные пакеты, поскольку принципы программирования Bluetooth-устройств одни и те же, независимо от программного обеспечения, которое вы применяете).

Для программирования Bluetooth-устройства на языке Java (впрочем, как и на любом другом языке) необходимо выполнить следующие действия:



- инициализацию стека;
- поиск устройства;
- настройку устройства;
- поиск сервисов устройства;
- обмен данными.

Перед тем, как что-либо делать с устройством, следует инициализировать стек. Вспомним, что стек представляет собой фрагмент программного или аппаратно-программного обеспечения, которое управляет устройством Bluetooth. Инициализация стека может включать несколько операций, но главная задача – подготовить устройство к установке беспроводного соединения и обмену данными. Вот исходный текст фрагмента программы, в котором выполняется инициализация Bluetooth-устройства:

```
import javax.bluetooth.*;
import javax.microedition.io.*;
import com.atinav.BCC;
public class WirelessDevice implements DiscoveryListener
 {

  LocalDevice localDevice = null;
  public WirelessDevice (){

  //установка используемого номера порта

  BCC.setPortName("COM1");

  //установка скорости обмена через COM–порт

  BCC.setBaudRate(57600);

  //установить режим соединения

  BCC.setConnectable(true);

  //установить режим поиска устройств

  BCC.setDiscoverable(DiscoveryAgent.GIAC);
  try
   {
    localDevice = LocalDevice.getLocalDevice();
   }
  catch (BluetoothStateException exp) {
 }

 // реализация методов в классе DiscoveryListener class
 }
}
```



Настройка устройства

LocalDevice и RemoteDevice — это два основных класса в Java Bluetooth Specification, позволяющих выполнить настройку устройства. С помощью этих классов можно получить статистическую информацию о вашем Bluetooth-устройстве (LocalDevice), а также информацию о других устройствах в сети (RemoteDevice). Статический метод LocalDevice.getLocalDevice() возвращает объект LocalDevice для дальнейшего использования в программе. Чтобы получить уникальный адрес вашего устройства, необходимо вызвать метод getBluetoothAddress() объекта LocalDevice. Адрес Bluetooth-устройства здесь имеет тот же смысл, что и MAC-адрес сетевой карты вашего компьютера, при этом каждое устройство Bluetooth имеет уникальный адрес. Если вам необходимо выполнить поиск других устройств в сети, то следует воспользоваться методом setDiscoverable() объекта LocalDevice.

Собственно говоря, это все, что необходимо сделать для настройки устройства Bluetooth при использовании пакета Java Bluetooth Specification APIs.

Сейчас посмотрим, как выполняется поиск других устройств Bluetooth.

Поиск устройств

Ваше устройство Bluetooth не знает, имеются ли другие Bluetooth-устройства в вашей сети. Для того чтобы обнаружить эти устройства, устройство Bluetooth должно использовать классы, имеющиеся в Java Bluetooth API. Следующие два класса позволяют вашему устройству обнаружить другие Bluetooth-устройства. Это DiscoveryAgent и DiscoveryListener.

После того как вы получили описатель вашего устройства LocalDevice, необходимо получить объект DiscoveryAgent, вызвав метод LocalDevice.getDiscoveryAgent():

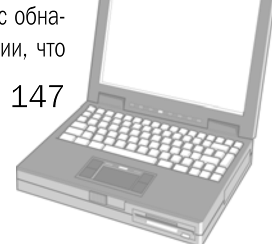
```
LocalDevice localdevice = LocalDevice.getLocalDevice();
DiscoveryAgent discoveryagent = localdevice.getDiscoveryAgent();
```

Существует масса способов, как обнаружить удаленные устройства Bluetooth, но здесь мы рассмотрим один. Первое, ваш объект должен поддерживать интерфейс DiscoveryListener. Этот интерфейс работает подобно любому другому прослушивающему интерфейсу (вспомните, например, реализацию TCP/IP интерфейса в сервере). При обнаружении запроса на соединение, генерируется событие, которое сообщает вам, что обнаружены какие-либо устройства в сети.

Чтобы начать процесс поиска, необходимо вызвать метод startInquiry() объекта DiscoveryAgent. Этот метод является неблокирующим, поэтому ваша программа может выполнять другие действия, пока идет поиск других Bluetooth-устройств. Когда удаленное устройство обнаружено, виртуальная Java-машина вызывает метод deviceDiscovered(), который реализует интерфейс DiscoveryListener. Этот метод возвращает объект RemoteDevice, который представляет обнаруженное в сети устройство.

Поиск сервиса

Когда вы обнаружили удаленное Bluetooth-устройство, желательно знать, какие сервисы (услуги) оно может обеспечить. Для получения такой информации следует воспользоваться методом searchServices() класса DiscoveryAgent, который выполняет поиск сервисов на удаленном устройстве, представленном объектом RemoteDevice. Если какой-либо сервис обнаружен, то виртуальная Java-машина вызывает метод servicesDiscovered() (при условии, что



в объекте был реализован интерфейс DiscoveryListener). Этот метод передает объект ServiceRecord, принадлежащий обнаруженному сервису. Получив этот объект, вы сможете работать с удаленным устройством, но для этого необходимо вначале установить связь с устройством, передавшим объект ServiceRecord.

Для установки связи с удаленным устройством вызовите метод getConnectionURL объекта ServiceRecord. Например, если получен один из объектов списка сервисов servRecord[i], то следующий оператор устанавливает связь с удаленным устройством:

```
String connectionURL = servRecord[i].getConnectionURL(0, false);
```

Регистрация сервиса

Перед тем как выполнять поиск сервисов удаленных Bluetooth-устройств, сервер Bluetooth должен зарегистрировать сервисы во внутренней базе данных под названием SDDB (Service Discovery DataBase). Этот процесс называется регистрацией сервиса. Здесь хочу напомнить, что в приложениях типа «точка-точка», например, ftp, telnet и др., любое устройство может быть либо клиентом, либо сервером. В этом случае для улучшения функциональности приложения желательно включить в него программный код, отрабатывающий как клиентский (например, поиск устройств), так и серверный (регистрация сервиса).

Ниже показан возможный алгоритм регистрации сервиса и сохранения данных в базе данных SDDB:

- вызовите метод Connector.open() и передайте результат интерфейсу StreamConnectionNotifier (здесь метод Connector.open() создает новый объект ServiceRecord и устанавливает для него некоторые атрибуты);
- используйте объекты LocalDevice и StreamConnectionNotifier для получения объекта ServiceRecord, созданного системой;
- если необходимо, модифицируйте или добавьте свои атрибуты в объект ServiceRecord;
- используйте StreamConnectionNotifier и вызов acceptAndOpen(), после чего ожидайте, пока сервисы не будут обнаружены клиентскими Bluetooth-устройствами и не будет выполнена установка соединения;
- если сервер должен отключиться, вызовите метод close() объекта StreamConnectionNotifier.

Классы StreamConnectionNotifier и Connector включены в пакет javax.microedition.io платформы J2ME platform. Ниже показан программный код, реализующий описанный алгоритм:

```
// инициализация переменных

StreamConnectionNotifier notifier = null;
StreamConnection sconn = null;
LocalDevice localdevice = null;
ServiceRecord servicerecord = null;

// шаг 1
// строка типа String с URL должна быть инициализирована

notifier = (StreamConnectionNotifier)Connector.open(url);
```

```
// шаг 2
// получим объект LocalDevice, если это еще не сделано

localdevice = LocalDevice.getLocalDevice();
servicerecord = localdevice.getRecord(notifier);

// шаг 3 (необязательный)
// шаг 4
// на этом шаге блокируется клиентский поток, пока не будет
// получен ответ от клиента. После того как ответ получен,
// запись о сервисе будет записана в SDDB

notifier.acceptAndOpen();

// шаг 5
// выполняется ожидание
// предположим, что клиент успешно соединился, тогда можно выходить
// шаг 6
// здесь запись сервиса может быть удалена из SDDB
notifier.close();
```

Это все, что необходимо для регистрации сервиса Bluetooth. Далее следует установить соединение с удаленным устройством.

Соединение и обмен данными

Для установления соединения и обмена данными пакет Java Bluetooth API предлагает три способа, но мы воспользуемся только одним из них, в котором используется протокол RFCOMM. Напомню, что RFCOMM представляет собой протокол «виртуального» последовательного порта, который применяется в профиле Serial Port Profile для организации соединения и обмена данными. Далее показан фрагмент программного кода, в котором демонстрируется, как открыть соединение с Bluetooth-устройством, работающим в режиме сервера:

```
// установим наши переменные

StreamConnectionNotifier notifier = null;
StreamConnection con = null;
LocalDevice localdevice = null;
ServiceRecord servicerecord = null;
InputStream input;
OutputStream output;

// инициализируем наш URL, который содержит уникальный идентификатор (UUID),
// что практически исключает возможность конфликта с другими устройствами

String url =
"btspp://localhost:00112233445566778899AABBCCDDEEFF;name=serialconn";

// устанавливаем соединение с указанным URL и
// передаем результат в StreamConnectionNotifier
```

```
notifier = (StreamConnectionNotifier)Connector.open(url);

// блокируем текущий поток, пока клиент не ответит

con = notifier.acceptAndOpen();

// ответ получен, открываем потоки ввода и вывода данных

input = con.openInputStream();
output = con.openOutputStream();

// отправляем данные через потоки ввода—вывода
. . .
```

Большей частью этот код напоминает тот, что был рассмотрен при выполнении регистрации сервиса. Переменная String url начинается с btspp://localhost: – это нужно при использовании профиля Serial Port Profile. Далее следует идентификатор UUID, равный 00112233445566778899AABBCCDDEEFF. Это обычный идентификатор (можно выбрать другой), наконец, последняя часть URL – имя сервиса (можно его не задавать). Если имя сервиса задано, то в базе данных SDDB запись о сервисе будет содержать строку:

```
ServiceName = serialconn
```

В данной реализации сервису назначается идентификатор канала, который клиент должен предоставить серверу для установления соединения.

Установка соединения для профиля Serial Port Profile для клиента J2ME выполняется очень просто:

```
Connector.open().
StreamConnection con =(StreamConnection)Connector.open(url);
```

Программа получает строку с URL, который нужен для установки соединения устройства Bluetooth с сервисом, полученным в объекте ServiceRecord при поиске сервиса. Вот фрагмент кода, показывающий установку соединения клиента с профилем Serial Port Profile и сервера с тем же профилем:

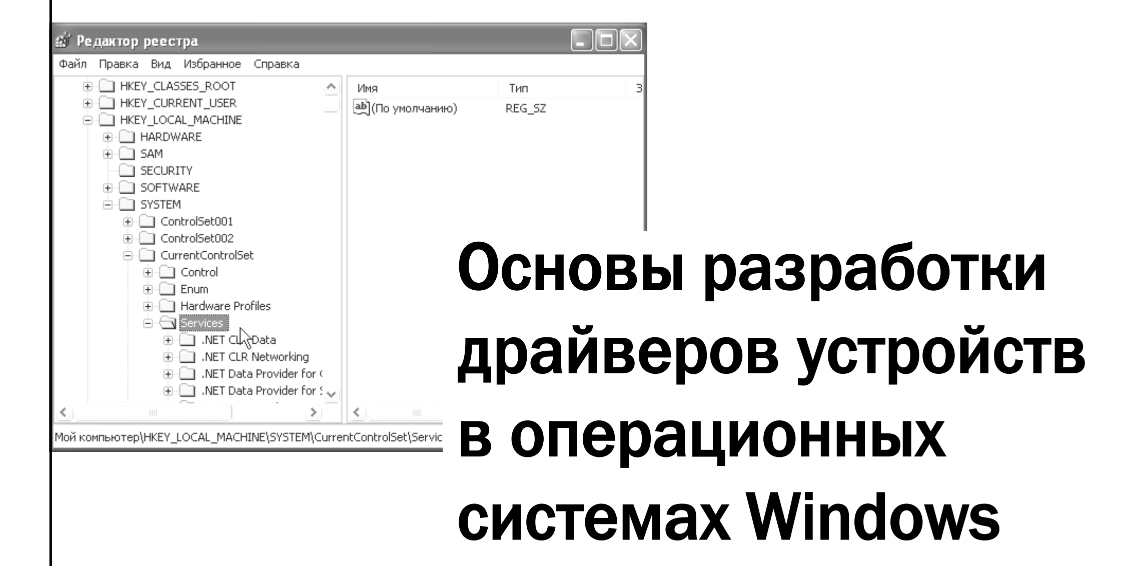
```
String connectionURL = serviceRecord.getConnectionURL(0, false);
StreamConnection con =(StreamConnection)Connector.open(connectionURL);
```

Как выглядит URL для клиента с профилем Serial Port Profile? Если, например, адрес сервера равен 0001234567AB, то строка с URL будет выглядеть, возможно, так:

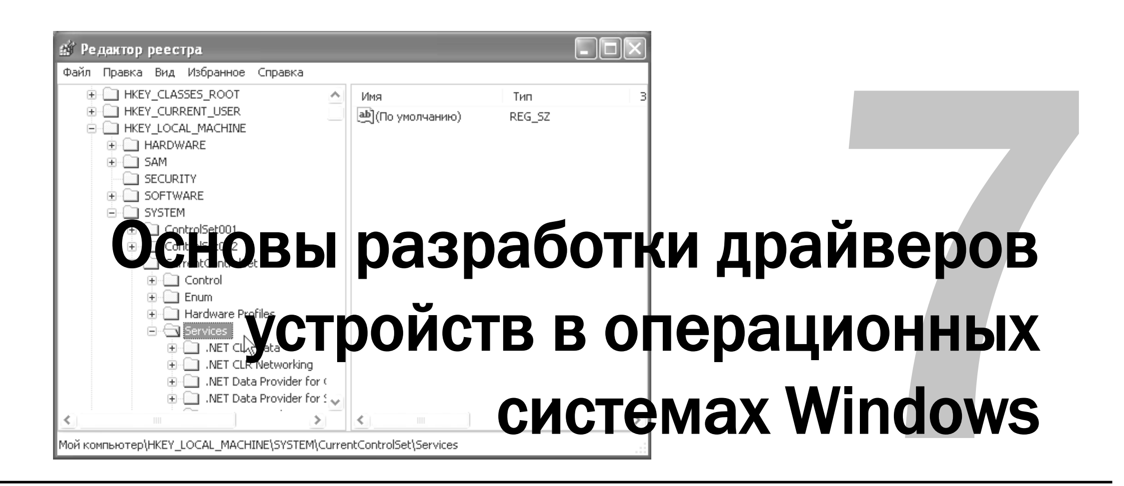
```
btspp://0001234567AB:3
```

Цифра 3 в конце записи указывает номер канала, присвоенный сервером сервису, когда сервис был прописан в базе SDDB.

В примерах программного кода мы использовали очень мощный пакет JSR–82 Development Kit фирмы Atinav, хотя можно использовать и очень популярный пакет J2ME Wireless Toolkit фирмы Sun Microsystems. Несмотря на кажущуюся сложность программирования Bluetooth-устройств, эти программные пакеты значительно упрощают этот процесс, что позволяет научиться программировать эти устройства даже новичкам.



Основы разработки драйверов устройств в операционных системах Windows



7 Основы разработки драйверов устройств в операционных системах Windows

В предыдущих главах этой книги для управления различными устройствами мы использовали свободно распространяемое программное обеспечение, куда входил драйвер PortTalk.sys и программа allowio.exe. В этой главе мы рассмотрим вопросы создания собственных драйверов устройств в операционных системах Windows, а также создадим драйвер для управления аналого-цифровым преобразователем, подключенным к параллельному порту ПК.

Разработка драйверов для Windows (как, впрочем, и для других операционных систем) окутана неким мистическим ореолом таинственности и считается чем-то из разряда «черной магии». Большинство программистов и пользователей полагают, что это чрезвычайно трудное занятие и доступно далеко не каждому. В определенном смысле это действительно так, если вы собираетесь разработать, например, драйвер звуковой или видеокарты.

Тем не менее, разработать несложный драйвер для обычного устройства из разряда домашней или лабораторной электроники может даже пользователь средней руки, достаточно хорошо владеющий основами программирования на языке С (поскольку все драйверы разрабатываются с использованием языка С) и имеющий хорошее представление о принципах организации операционных систем Windows. При разработке драйверов следует всегда помнить, что они работают в режиме ядра и при неправильном функционировании мгновенно могут привести операционную систему к краху («синий экран смерти»).

Напомню, что драйверу ядра доступны все аппаратные и программные ресурсы операционной системы, которые могут быть разрушены неправильно работающим программным кодом. Если вы захотите модифицировать приведенный в этой главе программный код драйверов или написать собственный драйвер, то тщательно изучите документацию по разработке драйверов, дабы избежать беспощадного разрушения системы.

Для разработки драйверов мы будем применять стандартное свободно распространяемое программное средство, выпущенное фирмой Microsoft – Windows DDK (Driver Development Kit). Этот программный пакет можно закачать с сайта Microsoft. В его состав включена обширная документация вместе с примерами разработок драйверов.

Сразу хочу оговориться, что мы не будем вникать во все аспекты разработки драйверов, поскольку это очень обширная и сложная тема, да это и не нужно. Для большинства устройств пользователя, описанных в этой книге, и многих других, которые несложно разработать, достаточно иметь самые простые драйверы устройств, в которых используются только основные приемы техники программирования драйверов. Все аспекты создания и функционирования драйверов мы будем рассматривать применительно к операционным системам Windows 2000/XP/2003/Vista.



Начнем с того, что определим, что собой представляет драйвер устройства, и как он взаимодействует с операционной системой. В упрощенном виде такую схему взаимодействия можно представить следующей схемой (рис. 7.1):

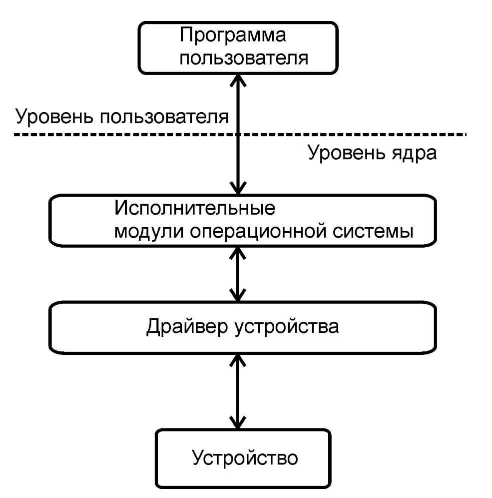
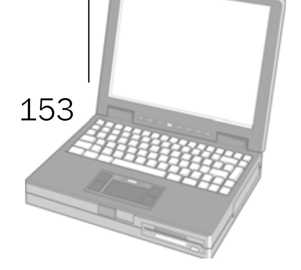


Рис. 7.1
Взаимодействие программы пользователя с устройством в операционной системе Windows.

В операционных системах Windows 2000/XP/2003/Vista программы пользователя взаимодействуют с любым физическим устройством (принтером, сканером и т. д.) посредством специальных программ – драйверов устройств. В этом случае программа пользователя, работающая в защищенных операционных системах, перечисленных выше, вначале обращается к исполнительным модулям Windows, представляющим собой программный интерфейс между пользователем и системой. Исполнительный модуль формирует запрос к драйверу устройства, который и выполняет операции с устройством. Такая иерархическая структура взаимодействия программы пользователя и устройства призвана обеспечить устойчивое функционирование операционной системы, которое не может быть нарушено какими-то операциями со стороны программы пользователя.

Такая иерархия в операционных системах Windows реализуется посредством присвоения привилегий любой работающей программе, или по-другому, процессу. Уровень привилегий определяет, может ли работающая программа (процесс) обращаться напрямую к системным ресурсам, таким, как физические адреса памяти, порты ввода-вывода, линии прерывания, устройства прямого доступа к памяти и т. д. Для программ пользователя, какими являются обычные приложения, доступ к ресурсам системы запрещен. Посмотрим на следующий пример. Попробуем из обычного консольного приложения, написанного в Delphi, обратиться к регистру 0x378 параллельного порта в Windows XP, написав такой код:

```
program Project1;
{$APPTYPE CONSOLE}
uses
  SysUtils;
begin
  try
    { TODO -oUser -cConsole Main : Insert code here }
    asm
```

```
      mov DX, 378h
      mov AL, 1h
      out DX, AL
    end;
  except
    on E:Exception do
      Writeln(E.Classname, ': ', E.Message);
  end;
end.
```

При запуске такого консольного приложения на дисплей будет выведено окно сообщения (рис. 7.2):

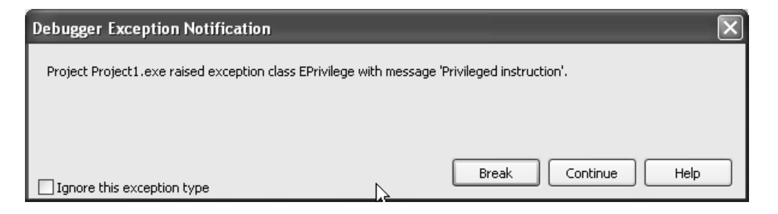


Рис. 7.2
Ошибка программы при попытке записи в параллельный порт

Команды ассемблера in и out являются «привилегированными» инструкциями процессора Intel и могут выполняться только программами, работающими на уровне ядра системы, поэтому пользовательская программа и завершается аварийным образом.

К ресурсам системы напрямую могут обращаться некоторые системные программы и большинство драйверов устройств. Для того чтобы пользовательская программа могла работать с привилегированными инструкциями процессора и обращаться напрямую к аппаратным ресурсам системы, нужно дать ей доступ на уровне ядра, сняв бит защиты в контексте процесса. Эти функции для примеров в предыдущих главах выполняли драйвер PortTalk.sys и программа allowio.exe.

Есть и другой путь для работы с устройствами пользователя – написать самому драйвер устройства и обращаться к нему из программы пользователя. Именно этот путь и будет здесь описан. Для того чтобы реализовать такую возможность нужно сделать два шага:

1) написать драйвер устройства;
2) написать программу, которая бы обращалась к драйверу устройства для выполнения операций ввода-вывода.

Прежде чем реализовать подобный проект, нужно изучить некоторые ключевые аспекты взаимодействия программы пользователя и драйвера устройства, а также базовые принципы разработки драйверов устройств.

7.1. Взаимодействие пользовательской программы с драйвером устройства

Приложение пользователя, работающее в Windows, может взаимодействовать с устройством через исполнительный уровень, на котором работают важные системные службы, посредством обращения к функциям интерфейса прикладного программирования (функции



WINAPI), которые включены в некоторые библиотеки динамической компоновки, например, в kernel32.dll. Здесь нужно отметить следующий важный момент: все устройства, с которыми взаимодействует операционная система, представлены в ней как файлы, причем это относится не только к файлам в их традиционном представлении как об определенным образом скомпонованным группам данных на жестком диске, но и ко всем аппаратным устройствам. Так, например, параллельный порт ПК в операционной системе Windows может быть представлен как файл с именем «LPT1», последовательному порту может быть присвоено имя «COM1» и т. д. Рассмотрим, например, как осуществляется взаимодействие программы пользователя с печатающим устройством, присоединенным к порту LPT1.

Это очень важный момент в нашем рассуждении для понимания всего последующего материала. Предположим, нам нужно напечатать из нашей программы символ на принтере. Схема взаимодействия приложения и устройства печати в упрощенном виде показана на рис. 7.3:

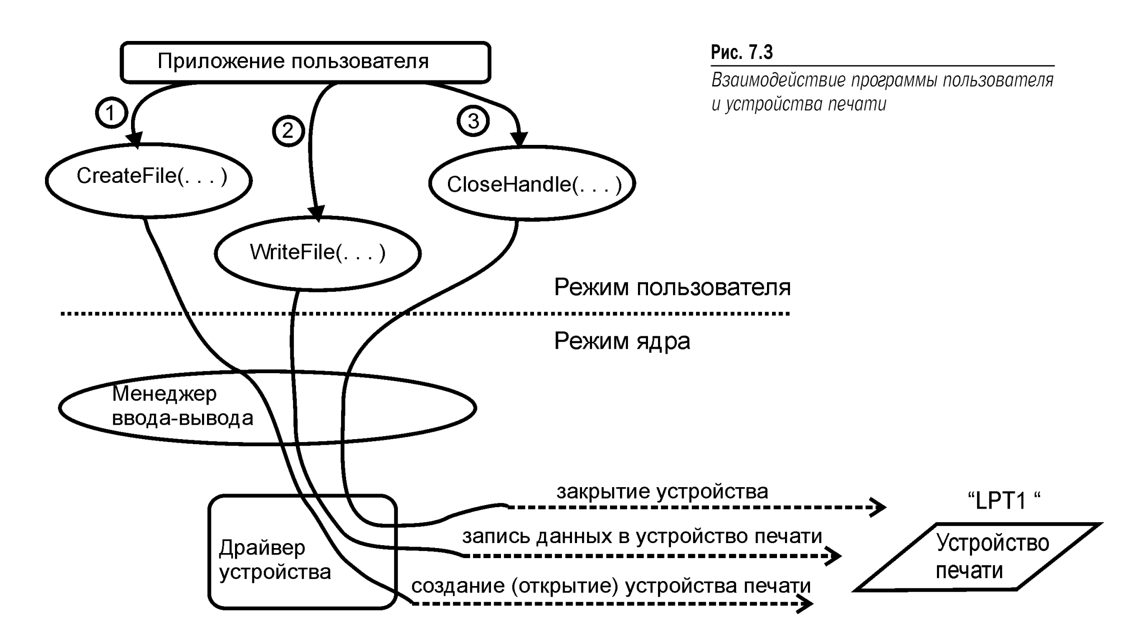


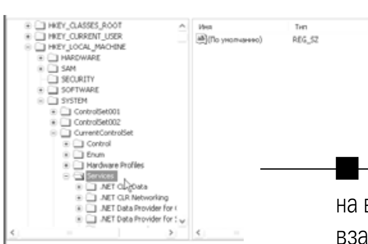
Рис. 7.3

Взаимодействие программы пользователя и устройства печати

Схема взаимодействия пользовательского приложения и устройства ввода-вывода (принтера) является весьма показательной и относится ко всем устройствам операционной системы. Первое, что должна сделать программа, перед тем как выполнять операции ввода-вывода с устройством – открыть его, что и выполняет системная функция CreateFile() (1). Эта функция всегда должна выполняться при обращении к объекту файловой системы, будь то дисковый файл или устройство ввода-вывода. В качестве первого параметра эта функция принимает имя устройства (в данном примере, это «LPT1») . Эта функция относится к прикладному интерфейсу программирования (так называемому интерфейсу WINAPI), который является своеобразным мостом между приложением пользователя и системой. Здесь есть один важный нюанс – функции WINAPI позволяют обратиться из программы, работающей в режиме пользователя, к системным функциям, работающим в режиме ядра.

В данном случае, функция WINAPI CreateFile() обращается к Менеджеру ввода-вывода операционной системы, который находится на исполнительном уровне и формирует запросы





на ввод-вывод в/из устройство(а). Менеджер ввода-вывода является важным звеном в цепи взаимодействия приложения и устройства ввода-вывода, осуществляя взаимодействие программы пользователя, с одной стороны, и устройства посредство его драйвера, с другой. Менеджер ввода-вывода – это обобщенное название системных сервисов и программ, осуществляющих формирование стандартных пакетов запросов (IRP) к драйверу устройства, буферизацию данных ввода-вывода устройства и синхронизацию очередей ввода-вывода.

При успешном завершении функции CreateFile() приложение получает дескриптор (описатель) открытого объекта файловой системы и все последующие операции с устройством осуществляются с использованием этого дескриптора. Драйвер устройства получает пакет запроса, соответствующий функции открытия файла и выполняет соответствующие действия по инициализации устройства (во многих случаях, при открытии-закрытии устройства никакой аппаратной инициализации не выполняется и драйвер просто возвращает статус успешного завершения).

На втором шаге программа должна записать байт данных в устройство печати. Это выполняется при помощи функции WINAPI WriteFile() (2). Как и в случае с функцией CreateFile(), цепочка обращений та же – Менеджер ввода-вывода–драйвер–устройство. Разница в том, что в этом случае Менеджер ввода-вывода операционной системы формирует пакет запроса записи данных с соответствующей инициализацией областей виртуальной памяти для передачи данных.

Наконец, после окончания работы с устройством следует закрыть его дескриптор, используя WINAPI функцию CloseHandle(3), которая закрывает сеанс работы с устройством, удаляя дескриптор и выделенные устройству ресурсы.

Это весьма упрощенный пример взаимодействия программы пользователя и устройства ввода-вывода, но все же он дает определенное представление о механизмах такого взаимодействия. Таким образом, для взаимодействия с устройством ввода-вывода программа пользователя так или иначе будет использовать функции WINAPI CreateFile(), WriteFile(), ReadFile() и CloseHandle() или их модифицированные версии. При этом последовательность операций в простейшем цикле обмена данными так:

1) функция CreateFile() открывает устройство ввода–вывода (оно должно к этому моменту существовать в системе). Результатом выполнения этой функции будет дескриптор устройства, с которым будут выполняться все операции обмена данными (чтение и запись);
2) с помощью функций ReadFile() и WriteFile() осуществляются необходимые операции (чтения/записи). Результатом работы этих функций является количество прочитанных (ReadFile()) или записанных (WriteFile()) байтов;
3) по окончанию операций обмена данными с устройством дескриптор устройства нужно закрыть функцией CloseHandle(). Таким образом, операционной системе сообщается, что устройство более не используется и выделенные ему аппаратно-программные ресурсы освобождаются.

Вместо функций ReadFile() и WriteFile() можно использовать WINAPI-функцию DeviceIoControl(). С помощью этой функции можно послать устройству код какой-либо операции, не только записи или чтения данных, но также любой другой код, на который устройство может реагировать. Это расширяет возможности операций ввода-вывода, но требует, чтобы драйвер устройства мог обрабатывать такие коды. В наших последующих примерах мы будем использовать именно функцию DeviceIoControl().



В принципе, перечисленных функций WINAPI вполне достаточно, чтобы ваше приложение могло работать с устройствами ввода-вывода. Посмотрим теперь, как создать сам драйвер устройства и как с ним работать, но вначале немного теории.

7.2. Основы функционирования драйверов в операционных системах Windows

Начнем с того, что в операционных системах Windows все физические устройства условно разделяются на две большие группы: поддерживающие технологию PnP или не поддерживающие ее. Соответственно, драйверы, поддерживающие технологию PnP, называют WDM-драйверами, а драйверы, не поддерживающие эту технологию – NT-драйверами. Это весьма упрощенный подход, но при первоначальном изучении данной темы этого достаточно. Оба типа драйверов в своей основе используют одни и те же функции, но WDM-драйвер включает ряд дополнительных функций, позволяющих реализовать возможности технологии PnP (установка-удаление без перезагрузки, управление энергопотреблением и т. д.). Кроме того, для установки и обновления WDM-драйвера устройства используется стандартный мастер Windows, знакомый всем пользователям (рис. 7.4):

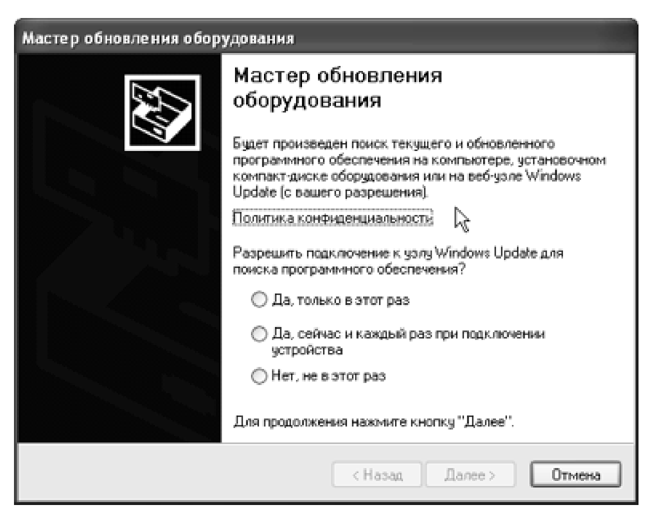
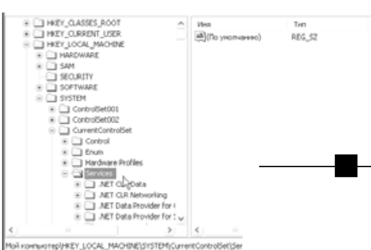


Рис. 7.4
Мастер обновления WDM-драйвера устройства

Для установки/обновления WDM-драйвера требуется наличие INF–файла, в котором описаны характеристики драйвера (имя файла, место установки, данные о производителе и т. д.).

Для установки NT-драйвера INF-файл не нужен, но требуется написать программу, позволяющую с помощью специальных функций WINAPI включить драйвер в систему. То же самое, в принципе, можно сделать и вручную, прописав соответствующие значения ключей в системном реестре. Этот тип драйвера используется для работы с устройствами не-PnP, а также при написании драйверов, не работающих с каким-либо оборудованием. В частности, NT-подобный драйвер можно использовать при работе с параллельным, последовательным портом, звуковой картой и другими устройствами в системе. Такой драйвер сравнительно несложно написать, что мы и продемонстрируем в последующих примерах.



Основы функционирования драйверов

Драйвер устройства операционной системы – это не обычная программа. Для пользователя, привыкшего к разработке обычных приложений, технология разработки драйвера покажется необычной. В драйвере нет точки входа в программу, как например, main в программе на языке С, хотя в момент начальной загрузки операционная система вызывает программу инициализации драйвера, известную под названием DriverEntry. Драйвер устройства можно представить как некий контейнер, в котором содержатся структуры данных и функции, что делает его в некотором смысле похожим на библиотеку динамической компоновки (dll). Обращение к драйверу при выполнении операций с устройством – это обращение к одной из его функций.

Это весьма упрощенное описание структуры драйвера, но оно дает представление о том, как работает драйвер устройства. Жизненный цикл функционирования драйвера устройства можно разделить (условно) на три этапа:

1) инициализацию;
2) выполнение операций по запросам операционной системы на ввод-вывод данных (Менеджер ввода-вывода);
3) остановка и удаление из системы.

Вначале драйвер устройства должен быть проинициализирован. Процесс инициализации в упрощенном виде показан на рис. 7.5:

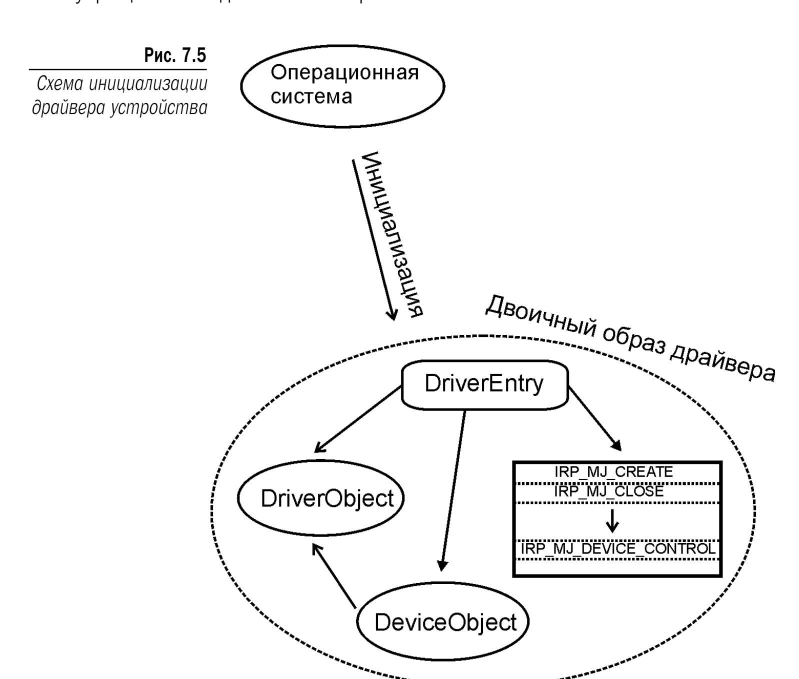


Рис. 7.5
Схема инициализации драйвера устройства

Драйвер устройства записан на диске в формате файла с расширением .sys. Для операционных систем Windows общепринято располагать файлы драйверов в катклоге windows\system32\drivers, хотя это вовсе не обязательно. Сведения об установленных драйверах хра-



нятся в системном реестре Windows в ключе \registry\machine\system\currentcontrolset\services. Исторически сложилось так, что записи о системных службах и драйверах хранятся в одном и том же разделе реестра (services). Помимо всего прочего записи содержат тип запуска драйвера – по нему система определяет, когда нужно загружать драйвер. Здесь есть очень важный момент: при испытаниях работоспособности драйвера желательно не инициализировать драйвер в момент загрузки системы, иначе можно вообще не загрузиться, если в драйвере имеется какая-либо ошибка.

При инициализации драйвера операционная система загружает двоичный образ дискового файла в защищенную область памяти и передает управление функции DriverEntry. Основная задача этой функции – проинициализировать поля структур DriverObject и DeviceObject и выполнить привязку устройства к пространству имен операционной системы, чтобы можно было обращаться к устройству из программы пользователя. Кроме этого, здесь же инициализируется массив указателей на функции вида IRP_MJ_XXX, где XXX обозначает операцию.

Например, указатель IRP_MJ_CREATE может ссылаться на функцию, которая будет вызываться при создании (открытии) устройства (например, функцией CreateFile() из программы пользователя). Каждый драйвер может включать различные функции, в зависимости от назначения устройства. Например, в устройство можно только записывать данные, но нельзя читать – в этом случае можно не определять функцию для IRP_MJ_READ или сделать ее в виде программной «заглушки».

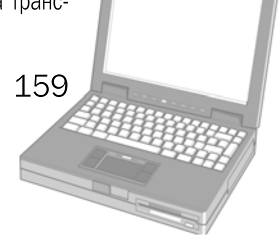
Как в драйвере выбираются функции, которые должны выполнять в данный момент операцию с устройством? Для этого предусмотрен механизм пакетов запроса ввода-вывода (I/O Request Packet, IRP). Например, если программа пользователя выполняет запись данных в устройство при помощи WINAPI-функции WriteFile(), то Менеджер ввода-вывода операционной системы формирует пакет запроса IRP_MJ_WRITE и посылает его драйверу устройства для выполнения. В свою очередь, драйвер устройства вызывает функцию, чей адрес указан при инициализации массива указателей и передает ей управление. Любой NT-драйвер, тем не менее, должен обязательно включать обработчики пакетов запроса IRP_MJ_CREATE (открытие устройства) и IRP_MJ_CLOSE (закрытие устройства).

Пакет запроса IRP содержит необходимые поля и адреса областей памяти, выделенных системой для временного хранения данных при чтении-записи. Мы не будем детально рассматривать структуру IRP, а перейдем к анализу функционирования драйвера.

Взаимодействие драйвера устройства с программой пользователя рассмотрим на примере чтения данных из устройства ввода-вывода (рис. 7.6).

На этом рисунке показана в упрощенном виде операция чтения данных устройства из приложения пользователя. Приложение пользователя вызывает функцию ReadFile(), передавая ей дескриптор открытого устройства. Затем Менеджер ввода-вывода, работающий в режиме ядра, по дескриптору устройства определяет, для какого драйвера следует формировать запрос IRP_MJ_READ и подготавливает пакет запроса. Далее вызывается функция обработки, соответствующая полученному запросу, которая выполняет необходимые действия по чтению данных из устройства. Здесь нужно отметить, что даже драйвер никогда не выполняет напрямую операций с портами ввода-вывода устройства, а передает управление уровню аппаратных абстракций, реализованному в виде библиотечных функций в библиотеке hal.dll. Фактически здесь «заканчивается» аппаратная «независимость» всех операций и операции проводятся на уровне конкретных аппаратных средств.

После выполнения чтения данных (успешного или нет) в драйвер устройства передаются данные (если есть) и статус выполненной операции. Далее Менеджер ввода-вывода трансформирует это в форму, с которой работает приложение и возвращает результат.



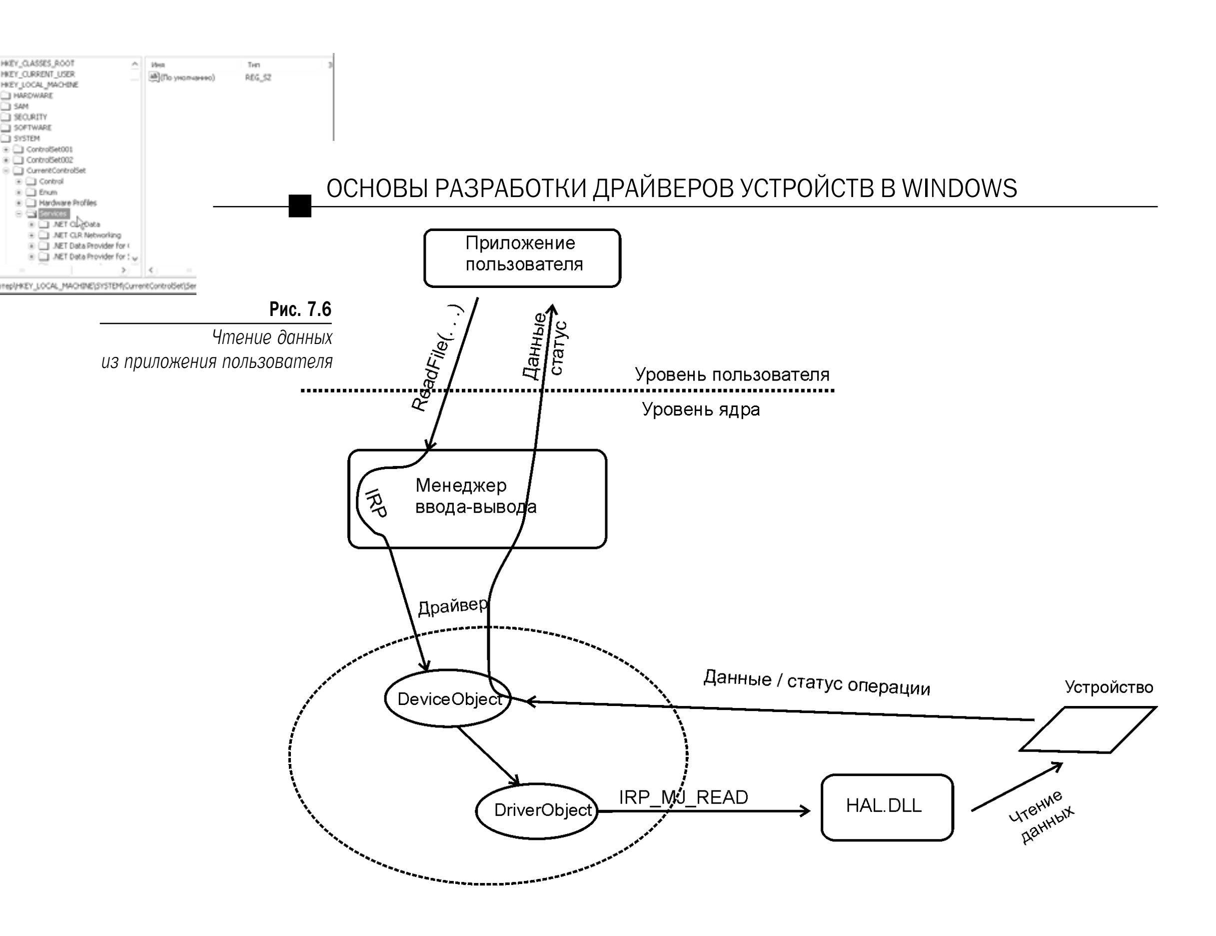


Рис. 7.6

Чтение данных из приложения пользователя

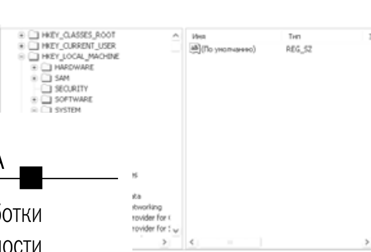
Таким образом, мы, в общем, рассмотрели схему работы NT-подобного драйвера устройства. Здесь не анализируются другие важнейшие аспекты функционирования драйверов (прерывания, потоки, синхронизация очередей и т. д.); заинтересованных читателей могу отослать к документации, входящей в состав Windows DDK.

Сейчас мы на практике посмотрим, как можно реализовать простые драйверы устройств и, чтобы почувствовать, как это работает, создадим простейший драйвер.

7.3. Разработка и отладка простейшего драйвера

Для того чтобы начать разрабатывать драйверы устройств Windows на компьютере должен быть установлен пакет Windows DDK (Driver Development Kit) фирмы Microsoft, желательно последней версии (на момент написания книги таковой являлась версия 2003 SP1). Этот пакет включает все необходимые средства для разработки и отладки драйверов. Кроме того, нужно иметь под рукой хороший компилятор C/C++ (подойдет бесплатная версия Microsoft Visual Studio Express Edition) для разработки приложения, используемого при тестировании драйвера. Поскольку для драйвера потребуется отладчик, то в качестве такого можно выбрать DebugView (www.microsoft.com), который, несмотря на свою простоту, довольно удобен в работе.

Этапы компиляции и сборки драйвера в Windows DDK довольно хорошо описаны во многих источниках, поэтому я не буду на этом останавливаться подробно. Давайте сразу перейдем к разработке нашего первого драйвера, который, вообще говоря, ничего не будет делать, а только выводить строки с сообщениями, которые можно будет перехватывать в нашем отладчике. Таким образом, мы посмотрим, как создать работоспособный драйвер и научимся



проверять его в отладчике DebugView. Работа с отладчиком – важнейший этап разработки драйвера, поскольку никаких других реальных способов обнаруживать неисправности в драйверах не существует.

В качестве первого примера разработаем драйвер «виртуального» устройства Test, который «ничего» не делает, а только выводит в окно отладчика текстовую строку о выполненной операции.

Вот исходный текст нашего простейшего драйвера:

```
#include <ntddk.h>

#define NT_DEVICE_NAME L"\\Device\\Test"
#define DOS_DEVICE_NAME L"\\DosDevices\\Test"

VOID
Test_Unload (IN PDRIVER_OBJECT DriverObject)
{
  UNICODE_STRING DosDeviceName;

  DbgPrint("%s","Test: UNLOADING!\n");
  RtlInitUnicodeString(&DosDeviceName, DOS_DEVICE_NAME);
  IoDeleteSymbolicLink(&DosDeviceName);
  IoDeleteDevice(DriverObject->DeviceObject);
}

NTSTATUS
Test_Create(IN PDEVICE_OBJECT DeviceObject,
           IN PIRP Irp)
{
  DbgPrint("%s", "Test: CREATED!\n");
  Irp->IoStatus.Status = STATUS_SUCCESS;
  IoCompleteRequest(Irp, IO_NO_INCREMENT);
  return STATUS_SUCCESS;
}

NTSTATUS
Test_Close(IN PDEVICE_OBJECT DeviceObject,
          IN PIRP Irp)
{
  DbgPrint("%s","Test: CLOSED!\n");
  Irp->IoStatus.Status = STATUS_SUCCESS;
  IoCompleteRequest(Irp, IO_NO_INCREMENT);
  return STATUS_SUCCESS;
}


NTSTATUS
DriverEntry(IN PDRIVER_OBJECT DriverObject,
           IN PUNICODE_STRING RegistryPath)
{
```

```
  PDEVICE_OBJECT DeviceObject;
  UNICODE_STRING NtDeviceName;
  UNICODE_STRING DosDeviceName;
  NTSTATUS status;

  RtlInitUnicodeString(&NtDeviceName, NT_DEVICE_NAME);
  status = IoCreateDevice(DriverObject,
                          0,
                          &NtDeviceName,
                          FILE_DEVICE_UNKNOWN,
                          0,
                          FALSE,
                          &DeviceObject);
  if (!NT_SUCCESS(status))
   {
    IoDeleteDevice(DeviceObject);
    return status;
   }

  RtlInitUnicodeString(&DosDeviceName, DOS_DEVICE_NAME);
  status = IoCreateSymbolicLink(&DosDeviceName, &NtDeviceName);

  if (!NT_SUCCESS(status))
   {
    IoDeleteSymbolicLink(&DosDeviceName);
    IoDeleteDevice(DeviceObject);
    return status;
   }

  DeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;
  DeviceObject->Flags |= DO_BUFFERED_IO;
  DriverObject->MajorFunction[IRP_MJ_CREATE]  = Test_Create;
  DriverObject->MajorFunction[IRP_MJ_CLOSE]   = Test_Close;
  DriverObject->DriverUnload = Test_Unload;

  return status;
}
```

В начало листинга нужно включить файл заголовка ntddk.h, в котором определены все необходимые функции. Далее мы определяем две строковые константы в формате UNICODE. Здесь мы должны уточнить некоторые нюансы. Строка NT_DEVICE_NAME указывает имя устройства, к которому будет обращаться наш драйвер, таким, каким оно должно быть в пространстве имен операционной системы. Здесь возникает проблема: приложение пользователя не может использовать функцию CreateFile() с данным именем, поскольку, в силу исторических обстоятельств, нужно указывать другой путь, определенный в константной строке DOS_DEVICE_NAME. Ничего трагического в этом нет, и мы может просто связать, или по-другому, сделать ссылку из DOS_DEVICE_NAME на NT_DEVICE_NAME, после чего все будет работать нормально.

Вспомним, что каждый драйвер обрабатывает только те пакеты запросов, которые программист для него определил. В нашем случае мы создали драйвер «виртуального» устрой-



ства Test, который обрабатывает только запросы IRP_MJ_CREATE, IRP_MJ_CLOSE и отдельно Unload. Функция Unload выгружает драйвер из системы и обрабатывается несколько иным способом, чем пакеты запросов (в WDM-драйверах эта функция вообще не используется).

Проанализируем исходный текст функции-обработчика IRP_MJ_CREATE:

```
NTSTATUS
Test_Create(IN PDEVICE_OBJECT DeviceObject,
           IN PIRP Irp)
{
  DbgPrint("%s", "Test: CREATED!\n");
  Irp->IoStatus.Status = STATUS_SUCCESS;
  IoCompleteRequest(Irp, IO_NO_INCREMENT);
  return STATUS_SUCCESS;
}
```

Первое – все функции-обработчики пакетов запросов принимают два параметра – адрес структуры DEVICE_OBJECT устройства и адрес пакета запроса PIRP. Оба параметра содержат поля, которые функция может использовать при обработке запроса.

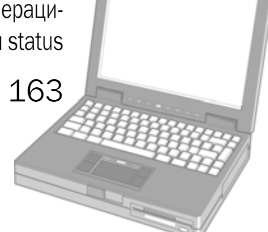
Функция DbgPrint – это специальная функция, предназначенная для вывода информации в отладчик. Эта функция используется только для трассировки программного кода и может быть перехвачена только в отладчике, таком, например, как DebugView. В функции Testdrv_Create также ничего существенного не делается. В поле Status пакета запроса заносится значение STATUS_SUCCESS, которое затем будет возвращено приложению как свидетельство успешно выполненной операции. Функция IoCompleteRequest завершает обработку пакета запроса. Сама функция возвращает значение STATUS_SUCCESS.

Точно так же работает и функция Testdrv_Close, которая, по запросу приложения (функция WINAPI CloseHandle()) закрывает дескриптор устройства.

Функция Testdrv_Unload удаляет символическую ссылку на устройство, о которой мы упоминали ранее (IoDeleteSymbolicLink()) и удаляет устройство из системы (IoDeleteDevice()).

Мы уже упоминали о функции DriverEntry, которая вызывается операционной системой при инициализации драйвера устройства. Функции передается ссылка на структуру DriverObject, созданной для нее операционной системой (первый параметр) и указатель на ключ данного драйвера в системном реестре (в записях этого ключа можно хранить или из записей извлекать при необходимости какие-то дополнительные параметры). Эта функция последовательно выполняет такие шаги (которые являются стандартными для этого класса драйверов):

1) инициализируется константная строка с именем устройства в пространстве имен системы (функция RtlInitUnicodeString(&NtDeviceName, NT_DEVICE_NAME);
2) создается программный объект описания устройства при помощи функции IoCreateDevice(). Если объект устройства создать не удалось, функция заканчивает работу с ошибкой, код которой передается в переменной status, а сам объект устройства удаляется;
3) инициализируется константная строка с именем устройства в пространстве имен, доступном пользовательской программе (функция RtlInitUnicodeString(&DosDeviceName, DOS_DEVICE_NAME));
4) создается символическая ссылка DOS-имени на имя из пространства имен операционной системы для доступа к устройству из приложения пользователя (функция status





= IoCreateSymbolicLink(&DosDeviceName, &NtDeviceName). В случае неудачи символическая ссылка и объект устройства удаляются, а функция DriverEntry заканчивается с ошибкой;
5) устанавливаются флаги устройства;
6) определяются функции-обработчики пакетов запроса.

Поместим исходный текст драйвера в файл test.c и сохраним его. Теперь в каталог, где находится файл, поместим еще два файла:

```
makefile
sources
```

С их помощью компилятор создаст (если нет ошибок в исходном тексте) файл драйвера устройства с именем test.sys. Смысл записей в этих файлах я объяснять не буду – это все есть в документации. Вы можете просто взять любую пару этих файлов из каталогов, куда помещены примеры, и изменить имя файла в sources следующим образом:

```
# The sources for the test device driver:
TARGETNAME=test
TARGETPATH=obj
TARGETTYPE=DRIVER
INCLUDES=..\
TARGETLIBS=  $(DDK_LIB_PATH)\wdmsec.lib
SOURCES=test.c
```

После этого выберите консоль для запуска компилятора в среде Windows DDK, например, **Windows XP Checked Build Environment** и перейдите в каталог, где находятся ваши рабочие файлы. Затем наберите команду

```
build —ceZ
```

Если в исходном тексте файла test.c нет ошибок, то файл будет откомпилирован успешно (рис. 7.7):

Рис. 7.7
Окно компиляции драйвера

```
Windows XP Checked Build Environment

H:\WINDDK\3790~1.183\src\Development\TEST>build -ceZ
BUILD: Adding /Y to COPYCMD so xcopy ops won't hang.
BUILD: Using 2 child processes
BUILD: Object root set to: ==> objchk_wxp_x86
BUILD: Compile and Link for i386
BUILD: Examining h:\winddk\3790~1.183\src\development\test directory for files t
o compile.
BUILD: Compiling (NoSync) h:\winddk\3790~1.183\src\development\test directory
1>Compiling - test.c for i386
BUILD: Compiling  h:\winddk\3790~1.183\src\development\test directory
BUILD: Linking h:\winddk\3790~1.183\src\development\test directory
1>Linking Executable - objchk_wxp_x86\i386\test.sys for i386
BUILD: Done

    2 files compiled
    1 executable built

H:\WINDDK\3790~1.183\src\Development\TEST>
```





Компилятор помещает файл драйвера test.sys в отдельный каталог. В нашем случае, это \objchk_wxp_x86\I386.

После компиляции нужно установить драйвер в систему и проверить его работу. Как это сделать? В отличие от обычного приложения драйвер должен устанавливаться в операционную систему специальным образом, так же как и удаляться из нее. Для удобства скопируем файл test.sys в корневой каталог какого-нибудь диска (в данном случае, это диск i:). Установим драйвер test.sys в системе, для чего разработаем простое консольное приложение в Visual C++ Express Edition, исходный текст которого показан далее:

```
#include <windows.h>
#include <stdio.h>

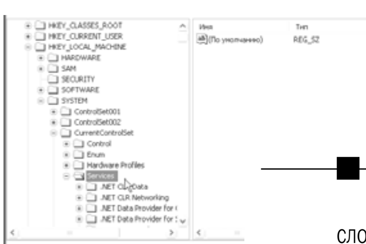

SC_HANDLE scm, svc;

void main(void)
{

  scm = OpenSCManager(NULL,
                      NULL,
                      SC_MANAGER_ALL_ACCESS);
  if (!scm)
   {
    printf("Cannot open SCM!\n");
    return;
   }

  svc = CreateService(scm,
                      "Test",
                      "Test",
                      SERVICE_ALL_ACCESS,
                      SERVICE_KERNEL_DRIVER,
                      SERVICE_DEMAND_START,
                      SERVICE_ERROR_NORMAL,
                      "i:\\test.sys",
                      NULL,
                      NULL,
                      NULL,
                      NULL,
                      NULL);
  if (!svc)
   {
    printf("Cannot open service!\n");
    CloseHandle(scm);
    return;
   }

  StartService(svc, 0, NULL);
  CloseServiceHandle(svc);
  CloseServiceHandle(scm);
}
```



Несмотря на небольшой объем исходного текста, программа может показаться довольно сложной. Проанализируем исходный текст. Вначале чуть-чуть теории. В операционных системах Windows имеется специальный механизм для установки, запуска, остановки и удаления системных служб (сервисов) и драйверов, реализованный посредством менеджера управления сервисами (Service Control Manager, SCM). Чтобы установить системный сервис или драйвер в операционную систему нужно вызвать целый ряд функций WINAPI. Мы не будем обсуждать детально механизм функционирования SCM, просто воспользуемся теми возможностями, которые он нам предоставляет.

Первоначально определим в программе две переменные – scm и svc типа SC_HANDLE (этот тип определяет дескрипторы для работы с SCM). Переменная scm будет содержать дескриптор базы данных SCM, а svc – дескриптор устанавливаемого драйвера.

Функция OpenSCManager обязательно вызывается первой. Она устанавливает связь с SCM, работающем на данном компьютере и открывает базу данных SCM, содержащую данные о системных службах и драйверах.

Если вызов функции OpenSCManager выполнен успешно, то следующим этапом будет установка нашего драйвера в системе, для чего служит функция CreateService() (ее параметры детально описаны в документации Microsoft). При успешном завершении этой функции в системе будет установлен драйвер нашего «виртуального» устройства test. Наконец, последний шаг – запустить наш драйвер, чтобы можно к нему обращаться из программы пользователя, что выполняет функция StartService(). После этого следует закрыть дескрипторы svc и scm.

В этой программе, с целью ее упрощения, не анализируются все ошибки, которые могут возникнуть при установке драйвера. Опять-таки, заинтересованных пользователей я отсылаю к документации Microsoft.

Проверить установлен ли драйвер в системе несложно. Для этого нужно из командной строки вызвать программу regedit, затем в системном реестре найти ключ \HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services (рис. 7.8):

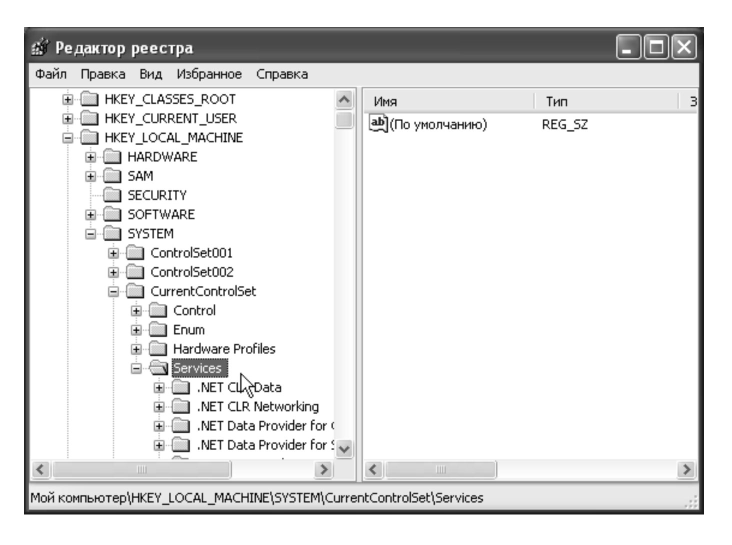
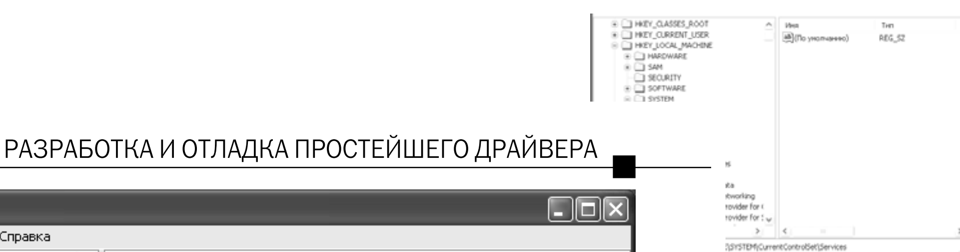
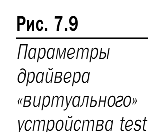


Рис. 7.8
Записи подключа Services системного реестра

В подключ Services системного реестра записываются параметры системных сервисов и драйверов устройств, установленных в системе. Для нашего «виртуального» устройства test в подключе Services после выполнения программы установки появится такая запись (рис. 7.9):





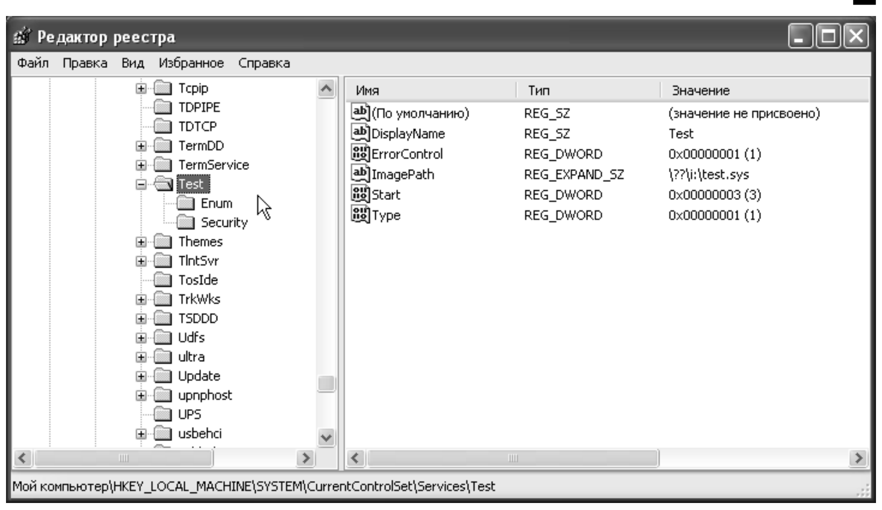


Рис. 7.9
Параметры драйвера «виртуального» устройства test

Параметры драйвера устройства, показанные на рис. 7.9, соответствуют тем, которые были заданы при установке драйвера функцией CreateService() нашей программы установки. Например, запись Type имеет целочисленное значение 1, что соответствует типу системного сервиса (SERVICE_KERNEL_DRIVER) функции CreateService(); запись Start имеет целочисленное значение 3, что соответствует ручному запуску (из приложения или операционной системы) (параметр SERVICE_DEMAND_START функции CreateService()).

Удалить драйвер устройства так же просто, как и установить. Вот исходный текст программы, которая удаляет драйвер «виртуального» устройства test из системы:

```
#include <windows.h>
#include <stdio.h>


SC_HANDLE scm, svc;
SERVICE_STATUS ServiceStatus;

void main(void)
{

  scm = OpenSCManager(NULL,
                    NULL,
                    SC_MANAGER_ALL_ACCESS);
  if (!scm)
  {
   printf("Cannot open SCM!\n");
   return;
  }

  svc = OpenService(scm, "Test", SERVICE_ALL_ACCESS);
  ControlService(svc, SERVICE_CONTROL_STOP, &ServiceStatus);
  DeleteService(svc);
```

```
  CloseServiceHandle(svc);
  CloseServiceHandle(scm);
}
```

В этом листинге присутствует уже знакомая нам функция OpenSCManager(), которая устанавливает связь с Service Control Manager и возвращает дескриптор базы данных системных сервисов. Предполагаем, что устройство test все еще присутствует в системе, и вызываем функцию OpenService() для получения доступа к нашему драйверу. Затем вызывается функция ControService(), назначение которой – остановить (если работает) драйвер или сервис. Опять-таки, здесь мы предполагаем, что запущенный нашей предыдущей программой драйвер работает, поэтому не проверяем возможные ошибки.

Здесь используется переменная ServiceStatus, в которую помещается код состояния сервиса или драйвера и константа SERVICE_CONTROL_STOP, которая используется для остановки драйвера. Наконец, функция DeleteService() удаляет наш драйвер из системы. Как обычно, перед выходом из программы следует закрыть открытые дескрипторы.

Если вы проверите после этого системный реестр, то увидите, что соответствующая запись из подключа Services удалена.

Таким образом, мы без проблем можем установить, запустить, остановить и удалить драйвер устройства из системы. Как мы можем теперь убедиться, что драйвер «виртуального» устройства test работает? Для этого мы можем воспользоваться свободно распространяемым фирмой Microsoft отладчиком DebugView. Использовать его очень просто, и мы сейчас в этом убедимся. Вспомним, что в исходном тексте нашего драйвера присутствуют функции DbgPrint, которые специально предназначены для вывода сообщений при отладке драйвера в окно отладчика. Таким образом, при обращении к устройству test из приложения мы сможем наблюдать в окне отладчика результаты обращений пользовательского приложения к устройству.

Нам нужно создать простое приложение на C++ (например, в Visual C++), которое бы устанавливать драйвер, открывать и закрывать «виртуальное» устройство test и удалять драйвер устройства. Мы только что рассмотрели два простых приложения, с помощью которых мы устанавливали и удаляли драйвер устройства. Скомбинировав исходные тексты этих приложений и добавив функции CreateFile() и CloseHandle(), мы получим приложение, результаты работы которого сможем наблюдать в отладчике DebugView. Вот исходный текст нашего приложения:

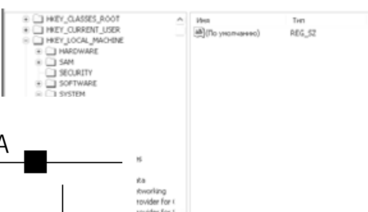
```
#include <windows.h>
#include <stdio.h>



HANDLE fh;
SC_HANDLE scm, svc;
SERVICE_STATUS ServiceStatus;

void main(void)
{

  scm = OpenSCManager(NULL,
                      NULL,
```




```
                  SC_MANAGER_ALL_ACCESS);
if (!scm)
 {
  printf("Cannot open SCM!\n");
  return;
 }

svc = CreateService(scm,
                 "Test",
                 "Test",
                 SERVICE_ALL_ACCESS,
                 SERVICE_KERNEL_DRIVER,
                 SERVICE_DEMAND_START,
                 SERVICE_ERROR_NORMAL,
                 "i:\\test.sys",
                 NULL,
                 NULL,
                 NULL,
                 NULL,
                 NULL);
if (!svc)
{
 printf("Cannot open service!\n");
 CloseHandle(scm);
 return;
}

StartService(svc, 0, NULL);
CloseServiceHandle(svc);
CloseServiceHandle(scm);


fh = CreateFile("\\\\.\\Test",
               GENERIC_READ | GENERIC_WRITE,
               0,
               NULL,
               OPEN_EXISTING,
               0,
               NULL);
CloseHandle(fh);


scm = OpenSCManager(NULL,
                  NULL,
                  SC_MANAGER_ALL_ACCESS);
if (!scm)
 {
  printf("Cannot open SCM!\n");
  return;
 }
```

```
  svc = OpenService(scm, "Test", SERVICE_ALL_ACCESS);
  ControlService(svc, SERVICE_CONTROL_STOP, &ServiceStatus);
  DeleteService(svc);

  CloseServiceHandle(svc);
  CloseServiceHandle(scm);

}
```

Эта программа будем устанавливать драйвер «виртуального» устройства test в системе, открывать устройство функцией CreateFile(), закрывать устройство функцией CloseHandle() и после этого удалять драйвер из системы. Таким образом, при работе этого приложения пользователя Менеджер ввода-вывода операционной системы будет формировать пакеты запросов IRP_MJ_CREATE, IRP_MJ_CLOSE, Unloading. При вызове соответствующих функций-обработчиков драйвера в программу-отладчик функцией DbgPrint будет выводиться соответствующая информация.

После компиляции приложения запустим отладчик DebugView и выполним наше приложение. В результате в окне отладчика появится такая информация (рис. 7.10):

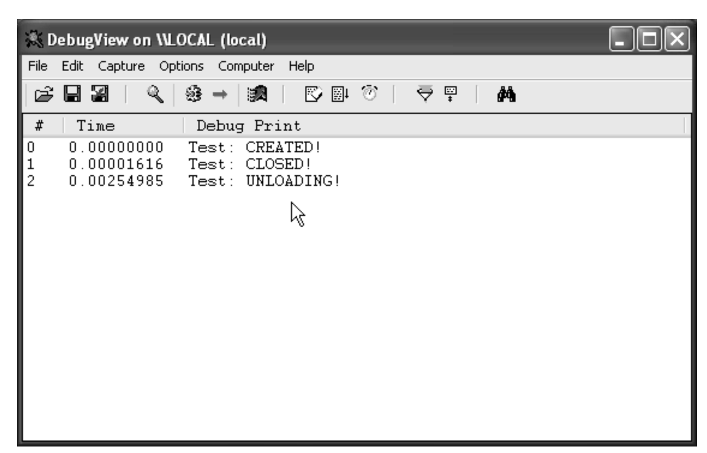


Рис. 7.10
Вид окна отладчика DebugView

Естественно, что для отладки драйвера можно использовать и другие отладчики, имеющиеся в Интернете.

Таким образом, мы знаем, как написать и отладить простейший, ничего не делающий драйвер. Теперь посмотрим, каким образом можно считывать или записывать данные в устройство ввода-вывода. Первое, что нужно определить при разработке такого драйвера – указать, в каких областях памяти должны находиться данные для записи-чтения. Чтение-запись данных довольно сложные процедуры с точки зрения операций, которые выполняет операционная система, но мы рассмотрим упрощенный вариант обмена данными между данными и приложением.

Для обмена данными мы использовать метод буферизации, когда данные, которые должны быть переданы приложению или которые должны быть прочитаны, приложением предварительно помещаются в специально выделенную системой область памяти (системный бу-



фер) и только затем обрабатываются приложением или драйвером. Далее, мы должны выделить для самого драйвера область памяти, где будут находиться прочитанные или записываемые данные. Для относительно небольшого объема данных можно воспользоваться областью расширения устройства (Device Extension) – областью памяти, которая может быть выделена драйверу при его создании и которая является неперемещаемой. Для того чтобы система могла выделить драйверу область памяти из расширения в исходный текст драйвера нужно внести некоторые изменения.

Первое – нужно явным образом определить структуру вместе с переменными, под которую и будет выделена область памяти в секции деклараций в начале исходного текста драйвера устройства. Вот пример:

```
typedef struct _MY_DEVICE_EXTENSION
{
      LONG L1;
} MY_DEVICE_EXTENSION, *PMY_DEVICE_EXTENSION;
```

Здесь определена структура с одной 32-битовой целочисленной переменной L1. Поскольку требуется всего 4 байта памяти, то система (гипотетически) могла бы выделить эти 4 байта, но, поскольку память выделяется блоками с учетом выравнивания для увеличения быстродействия, то, скорее всего, выделенный объем будет больше. Кроме того, при создании объекта устройства в функции IoCreateDevice в качестве второго параметра должен задаваться размер области расширения, например:

```
status = IoCreateDevice(DriverObject,
                        sizeof(MY_DEVICE_EXTENSION),
                       &NtDeviceName,
                       FILE_DEVICE_UNKNOWN,
                       0,
                       FALSE,
                       &DeviceObject);
```

Здесь вторым параметром функции IoCreateDevice является sizeof (MY_DEVICE_EXTENSION).

Какие функции используются для операций записи-чтения? Что касается приложения пользователя, то мы уже рассматривали этот вопрос. Напомню, что приложение может записать данные в дескриптор устройства с помощью функции WINAPI WriteFile(), а прочитать данные с помощью функции ReadFile(). Есть и универсальная функция DeviceIoControl(), позволяющая выполнять как чтение, так и запись данных.

Для манипуляций с данными в драйвере устройства можно использовать одну из функций ядра вида RtlXXX. Очень часто используется функция RtlMoveMemory(), которая перемещает данные из системной области памяти в область памяти драйвера (при записи данных в устройство) и наоборот, из области памяти драйвера в системный буфер (при чтении данных из устройства).

При обмене данными приложения и устройства обычно используются пакеты запроса IRP_MJ_READ, IRP_MJ_WRITE и IRP_MJ_DEVICE_CONTROL. При этом для драйвера должен быть указан тип обмена данными (буферизованный или небуферизованный) в функции DriverEntry. Во всех наших примерах мы будем использовать буферизованный обмен данны-



ми, поэтому после успешного создания объекта устройства функцией IoCreateDevice нужно указывать оператор

```
DeviceObject->Flags |= DO_BUFFERED_IO;
```

Еще один важный момент, касающийся операций ввода-вывода (чтения-записи): Нельзя пытаться выполнять какие-либо операции с устройством в функции DriverEntry! Она вызывается системой единственный раз при инициализации драйвера устройства и после выполнения будет выгружена из памяти! Все операции обмена данными в драйвере выполняются при поступлении пакетов запросов посредством соответствующих функций в массиве указателей IRP_MJ_XXX.

И, наконец, последнее перед тем, как перейти к практике – в наших последующих примерах мы будем для обмена данными использовать функцию DeviceIoControl, для которой Менеджер ввода-вывода операционной системы формирует пакет IRP_MJ_DEVICE_CONTROL.

7.4. Чтение-запись данных

В нашем первом примере прочитаем данные из устройства и выведем их значение на экран. Для этого нам понадобится разработать драйвер устройства и приложение, которое будет читать данные. Начнем с приложения. Наше приложение будет устанавливать и запускать драйвер, затем читать из него данные, после чего будет останавливать драйвер и выгружать его из системы. Несмотря на кажущуюся сложность такого приложения, большая его часть уже анализировалась нами ранее (установка и удаление драйвера), поэтому нам останется добавить программный код для чтения данных из устройства. Вот исходный текст этой программы:

```
#include <windows.h>
#include <stdio.h>

#define IOCTL_READ CTL_CODE(FILE_DEVICE_UNKNOWN,\
                           0x801,\
                          METHOD_BUFFERED,\
                          FILE_ANY_ACCESS)

SC_HANDLE scm, svc;
SERVICE_STATUS ServiceStatus;

void main(void)
{
  HANDLE fh;
  int i1;
  DWORD bytes;

  scm = OpenSCManager(NULL,
                    NULL,
                    SC_MANAGER_ALL_ACCESS);
  if (!scm)
```




```
 {
  printf("Cannot open SCM!\n");
  return;
 }

svc = CreateService(scm,
                  "Testr",
                  "Testr",
                  SERVICE_ALL_ACCESS,
                  SERVICE_KERNEL_DRIVER,
                  SERVICE_DEMAND_START,
                  SERVICE_ERROR_NORMAL,
                  "i:\\testr.sys",
                  NULL,
                  NULL,
                  NULL,
                  NULL,
                  NULL);
if (!svc)
 {
  printf("Cannot open service!\n");
  CloseHandle(scm);
  return;
 }

StartService(svc, 0, NULL);
CloseServiceHandle(svc);
CloseServiceHandle(scm);


fh = CreateFile("\\\\.\\Testr",
               GENERIC_READ | GENERIC_WRITE,
               0,
               NULL,
               OPEN_EXISTING,
               0,
               NULL);
if (fh != INVALID_HANDLE_VALUE)
 {
  DeviceIoControl(fh,
                IOCTL_READ,
                NULL,
                0,
                &i1,
                sizeof(i1),
                &bytes,
                NULL);
  printf("Testr: IOCTL_READ = %d\n", i1);
 }
```

```
  CloseHandle(fh);


  scm = OpenSCManager(NULL,
                    NULL,
                     SC_MANAGER_ALL_ACCESS);
  if (!scm)
   {
    printf("Cannot open SCM!\n");
    return;
   }
  svc = OpenService(scm, "Testr", SERVICE_ALL_ACCESS);
  ControlService(svc, SERVICE_CONTROL_STOP, &ServiceStatus);
  DeleteService(svc);

  CloseServiceHandle(svc);
  CloseServiceHandle(scm);

}
```

Наша программа будет взаимодействовать с «виртуальным» устройством testr, драйвер для которого мы разработаем позже, а сейчас остановимся более детально на функции WINAPI DeviceIoControl(), с помощью которой мы прочитаем данные из нашего устройства.

Прежде всего, для функции DeviceIoControl() нужно определить коды команд, которые должны выполняться. В нашем случае будет выполняться только чтение данных из устройства, поэтому используется одна команда, которой можем присвоить имя IOCTL_READ:

```
#define  IOCTL_READ CTL_CODE(FILE_DEVICE_UNKNOWN,\
                              0x801,\
                            METHOD_BUFFERED,\
                            FILE_ANY_ACCESS)
```

Команда формируется при помощи макроса CTL_CODE и ее второй параметр должен превышать число 0x800 (пользовательский диапазон). Коды IOCTL-команд, а также их атрибуты в приложении и в драйвере должны совпадать.

Функция DeviceIoControl в качестве параметров, кроме всего прочего, принимает адрес и размер входного и выходного буферов данных. При этом «входной» буфер функции содержит данные для записи в устройство, а «выходной» буфер – данные, принятые от устройства. В нашей программе данные будут только читаться, поэтому параметры входного буфера принимаются равными NULL и 0, а в выходной буфер i1 будет записано значение целочисленной переменной, считанной из устройства. Значение этой переменной затем будет выведено на экран дисплея.

Остальные фрагменты программного кода используются загрузки-выгрузки драйвера устройства и нами проанализированы ранее.

Перейдем теперь к драйверу устройства. Наше устройство называется testr и из него можно только читать данные. Исходный текст драйвера этого устройства показан далее:




```
#include <ntddk.h>

#define IOCTL_READ CTL_CODE(FILE_DEVICE_UNKNOWN,\
                            0x801,\
                          METHOD_BUFFERED,\
                          FILE_ANY_ACCESS)

#define NT_DEVICE_NAME L"\\Device\\Testr"
#define DOS_DEVICE_NAME L"\\DosDevices\\Testr"

typedef struct _MY_DEVICE_EXTENSION
{
  LONG L1;
} MY_DEVICE_EXTENSION, *PMY_DEVICE_EXTENSION;

VOID
Testr_Unload (IN PDRIVER_OBJECT DriverObject)
{
 UNICODE_STRING DosDeviceName;

 RtlInitUnicodeString(&DosDeviceName, DOS_DEVICE_NAME);
 IoDeleteSymbolicLink(&DosDeviceName);
 IoDeleteDevice(DriverObject->DeviceObject);
}

NTSTATUS
Testr_Create(IN PDEVICE_OBJECT DeviceObject,
             IN PIRP Irp)
{
 Irp->IoStatus.Status = STATUS_SUCCESS;
 IoCompleteRequest(Irp, IO_NO_INCREMENT);
 return STATUS_SUCCESS;
}

NTSTATUS
Testr_Close(IN PDEVICE_OBJECT DeviceObject,
            IN PIRP Irp)
{
 Irp->IoStatus.Status = STATUS_SUCCESS;
 IoCompleteRequest(Irp, IO_NO_INCREMENT);
 return STATUS_SUCCESS;
}

NTSTATUS
Testr_IoctlR(IN PDEVICE_OBJECT DeviceObject,
             IN PIRP Irp)
{
 PMY_DEVICE_EXTENSION dx=(PMY_DEVICE_EXTENSION)DeviceObject-
>DeviceExtension;
```

```
 PIO_STACK_LOCATION pIoStack = IoGetCurrentIrpStackLocation(Irp);

 ULONG ctlCode = pIoStack->Parameters.DeviceIoControl.IoControlCode;
 ULONG OutputLength = pIoStack-
>Parameters.DeviceIoControl.OutputBufferLength;

 PUCHAR buf = (PUCHAR)Irp->AssociatedIrp.SystemBuffer;

 dx->L1 = 337;
 OutputLength = 4;

 if (ctlCode == IOCTL_READ)
  {
   RtlMoveMemory(buf,
                (PUCHAR)&dx->L1,
               OutputLength);

  }
 Irp->IoStatus.Information = OutputLength;
 Irp->IoStatus.Status = STATUS_SUCCESS;
 IoCompleteRequest(Irp, IO_NO_INCREMENT);
 return STATUS_SUCCESS;
}

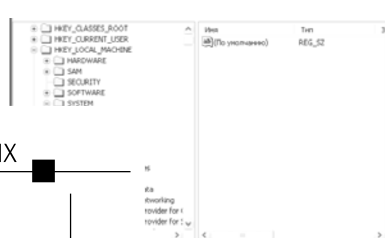
NTSTATUS
DriverEntry(IN PDRIVER_OBJECT DriverObject,
            IN PUNICODE_STRING RegistryPath)
{
 PDEVICE_OBJECT DeviceObject;
 UNICODE_STRING NtDeviceName;
 UNICODE_STRING DosDeviceName;
 NTSTATUS status;

 RtlInitUnicodeString(&NtDeviceName, NT_DEVICE_NAME);
 status = IoCreateDevice(DriverObject,
                         sizeof(MY_DEVICE_EXTENSION),
                        &NtDeviceName,
                        FILE_DEVICE_UNKNOWN,
                        0,
                        FALSE,
                        &DeviceObject);
 if (!NT_SUCCESS(status))
  {
   IoDeleteDevice(DeviceObject);
   return status;
  }

 RtlInitUnicodeString(&DosDeviceName, DOS_DEVICE_NAME);
 status = IoCreateSymbolicLink(&DosDeviceName, &NtDeviceName);

 if (!NT_SUCCESS(status))
```






```
 {
  IoDeleteSymbolicLink(&DosDeviceName);
  IoDeleteDevice(DeviceObject);
  return status;
 }

 DeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;
 DeviceObject->Flags |= DO_BUFFERED_IO;

 DriverObject->MajorFunction[IRP_MJ_CREATE]  = Testr_Create;
 DriverObject->MajorFunction[IRP_MJ_CLOSE]   = Testr_Close;
 DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = Testr_IoctlR;
 DriverObject->DriverUnload = Testr_Unload;

 return status;
}
```

Анализ исходного текста драйвера (назовем его testr.sys) начнем с раздела деклараций в начале листинга. Как и в приложении пользователя, только что нами рассмотренном, здесь присутствует объявление команды IOCTL_READ:

```
#define IOCTL_READ CTL_CODE(FILE_DEVICE_UNKNOWN,\
                           0x801,\
                          METHOD_BUFFERED,\
                          FILE_ANY_ACCESS)
```

Как видно, это объявление совпадает с тем, что указано в пользовательском приложении.

Для хранения данных нам нужна область памяти, для чего воспользуемся областью, выделяемой под расширение драйвера устройства (DeviceExtension). считываться будет всего одна целочисленная переменная, поэтому область расширения объявляется следующим образом:

```
typedef struct _MY_DEVICE_EXTENSION
{
  LONG L1;
} MY_DEVICE_EXTENSION, *PMY_DEVICE_EXTENSION;
```

В этой структуре указана 32-битовая целочисленная переменная L1, значение которой будет прочитано приложением.

В нашем драйвере будет выполняться чтение данных по IOCTL-команде, поэтому должен присутствовать обработчик запроса IRP_MJ_DEVICE_CONTROL, что и указывается в функции DriverEntry следующей строкой:

```
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = Testr_IoctlR;
```

В качестве обработчика запроса выступает функция Test_IoctlR. Проанализируем ее исходный текст. Прежде всего, создаем переменную-указатель dx на область расширения, в которой хранится значение переменной L1. Затем для удобства работы создаем перемен-



ные ctlCode и InputLength, которые будут содержать соответственно код команды IOCTL и размер передаваемых в приложение данных. Для передачи данных система создает буфер памяти, адрес которого мы присваиваем переменной buf.

Переменной L1, находящейся в области расширения, присваиваем произвольное значение (в данном случае, 337), которое и будет передано в приложение пользователя. Кроме того, нам потребуется указать точный размер (в байтах) передаваемой области данных. Эти действия выполняются двумя операторами:

```
dx->L1 = 337;
OutputLength = 4;
```

Далее анализируем код команды (оператор if) и, если это IOCTL_READ, то выполняем пересылку данных из области расширения драйвера в системный буфер с помощью функции RtlMoveMemory:

```
RtlMoveMemory(buf,
              (PUCHAR)&dx->L1,
             OutputLength);
```

В этой функции первый параметр – адрес области памяти, куда будут перемещены данные, второй параметр – адрес области памяти, откуда будут перемещаться данные, и, наконец, третий параметр показывает размер пересылаемых данных в байтах.

Перед завершением выполнения запроса функция должна передать Менеджеру ввода-вывода информацию о размере выводимых данных, что выполняет опреатор

```
Irp->IoStatus.Information = OutputLength;
```

Затем запрос завершается, как обычно, установкой статуса операции и вызовом функции IoCompleteRequest:

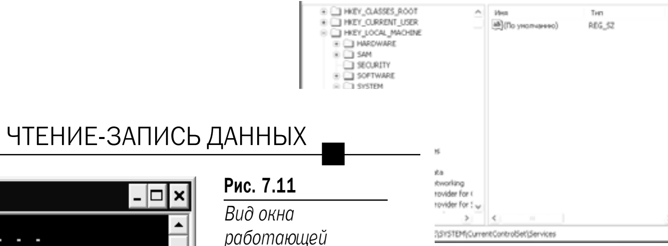
```
Irp->IoStatus.Status = STATUS_SUCCESS;
IoCompleteRequest(Irp,  IO_NO_INCREMENT);
```

Вот, собственно, и все об этом драйвере. Перед компиляцией исходного текста в Windows DDK не забудьте указать корректное имя в файле sources. Напомню, что сборки драйвера нужны два файла – makefile и sources. Оба можно взять из примеров, затем откорректировать sources, после чего выполнить компиляцию драйвера.

Скопируем файл драйвера testr.sys в корневой каталог i: и запустим пользовательское приложение. На экране дисплея увидим результат работы (рис. 7.11):

В рассмотренном нами примере приложение пользователя выполнило чтение данных из «виртуального» устройства testr. С таким же успехом можно и записывать данные в устройство. Сейчас мы рассмотрим более сложный пример, в котором пользовательское приложение записывает целочисленное значение в «виртуальное» устройство ввода-вывода, а затем читает из него значение, умноженное на 3. Вот исходный текст приложения:

```
#include <windows.h>
#include <stdio.h>
```




```
H:\WINDOWS\system32\cmd.exe
Testr: IOCTL_READ = 337
Для продолжения нажмите любую клавишу . . .
```

Рис. 7.11
Вид окна работающей программы чтения данных из устройства testr

```
#define IOCTL_READ CTL_CODE(FILE_DEVICE_UNKNOWN,\
                           0x801,\
                         METHOD_BUFFERED,\
                         FILE_ANY_ACCESS)
#define IOCTL_WRITE CTL_CODE(FILE_DEVICE_UNKNOWN,\
                            0x802,\
                          METHOD_BUFFERED,\
                          FILE_ANY_ACCESS)


SC_HANDLE scm, svc;
SERVICE_STATUS ServiceStatus;

void main(void)
{
  HANDLE fh;
  int inp;
  int outp = 233;

  DWORD bytes;

  scm = OpenSCManager(NULL,
                    NULL,
                    SC_MANAGER_ALL_ACCESS);
  if (!scm)
   {
    printf("Cannot open SCM!\n");
    return;
   }

  svc = CreateService(scm,
                    "Testw",
                    "Testw",
                    SERVICE_ALL_ACCESS,
                    SERVICE_KERNEL_DRIVER,
```

```
                    SERVICE_DEMAND_START,
                    SERVICE_ERROR_NORMAL,
                    "i:\\testw.sys",
                    NULL,
                    NULL,
                    NULL,
                    NULL,
                    NULL);
if (!svc)
 {
  printf("Cannot open service!\n");
  CloseHandle(scm);
  return;
 }

StartService(svc, 0, NULL);
CloseServiceHandle(svc);
CloseServiceHandle(scm);


fh = CreateFile("\\\\.\\Testw",
               GENERIC_READ | GENERIC_WRITE,
               0,
               NULL,
               OPEN_EXISTING,
               0,
               NULL);
if (fh != INVALID_HANDLE_VALUE)
 {
  DeviceIoControl(fh,
                 IOCTL_WRITE,
                 &outp,
                 sizeof(outp),
                 NULL,
                 0,
                 &bytes,
                 NULL);
  DeviceIoControl(fh,
                 IOCTL_READ,
                 NULL,
                 0,
                 &inp,
                 sizeof(inp),
                 &bytes,
                 NULL);
 printf("TESTW: output = %d\n", outp);
 printf("TESTW: input = output * 3 = %d\n", inp);
}
CloseHandle(fh);


scm = OpenSCManager(NULL,
```

```
                     NULL,
                     SC_MANAGER_ALL_ACCESS);
  if (!scm)
  {
   printf("Cannot open SCM!\n");
   return;
  }
  svc = OpenService(scm, "Testw", SERVICE_ALL_ACCESS);
  ControlService(svc, SERVICE_CONTROL_STOP, &ServiceStatus);
  DeleteService(svc);

  CloseServiceHandle(svc);
  CloseServiceHandle(scm);
}
```

В этом приложении будут выполняться две операции: вначале в устройство testw будет записано число 233 (переменная outp), а затем из устройства будет прочитано значение этого же числа, умноженное на 3 (переменная inp). Для записи в устройство используется команда IOCTL_WRITE с кодом 0x802, а для чтения из устройства применяется, как и в предыдущем примере, IOCTL-команда IOCTL_READ (код 0x801). Обе команды выполняет функция DeviceIoControl().

Исходный текст драйвера устройства (файл testw.sys), обрабатывающего запросы чтения-записи программы пользователя, также немного усложнился:

```
#include <ntddk.h>

#define IOCTL_READ CTL_CODE(FILE_DEVICE_UNKNOWN,\
                           0x801,\
                         METHOD_BUFFERED,\
                         FILE_ANY_ACCESS)

#define IOCTL_WRITE CTL_CODE(FILE_DEVICE_UNKNOWN,\
                            0x802,\
                          METHOD_BUFFERED,\
                          FILE_ANY_ACCESS)

#define NT_DEVICE_NAME L"\\Device\\Testw"
#define DOS_DEVICE_NAME L"\\DosDevices\\Testw"

typedef struct _MY_DEVICE_EXTENSION
{
      LONG L1;
} MY_DEVICE_EXTENSION, *PMY_DEVICE_EXTENSION;

VOID
Testw_Unload (IN PDRIVER_OBJECT DriverObject)
{
  UNICODE_STRING DosDeviceName;

  RtlInitUnicodeString(&DosDeviceName, DOS_DEVICE_NAME);
```




```
  IoDeleteSymbolicLink(&DosDeviceName);
  IoDeleteDevice(DriverObject->DeviceObject);
}

NTSTATUS
Testw_Create(IN PDEVICE_OBJECT DeviceObject,
             IN PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS;
  IoCompleteRequest(Irp, IO_NO_INCREMENT);
  return STATUS_SUCCESS;
}

NTSTATUS
Testw_Close(IN PDEVICE_OBJECT DeviceObject,
            IN PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS;
  IoCompleteRequest(Irp, IO_NO_INCREMENT);
  return STATUS_SUCCESS;
}

NTSTATUS
Testw_IoctlRW(IN PDEVICE_OBJECT DeviceObject,
              IN PIRP Irp)
{
  PMY_DEVICE_EXTENSION dx = (PMY_DEVICE_EXTENSION)DeviceObject-
>DeviceExtension;
  PIO_STACK_LOCATION pIoStack = IoGetCurrentIrpStackLocation(Irp);

  ULONG ctlCode = pIoStack->Parameters.DeviceIoControl.IoControlCode;
  ULONG OutputLength = pIoStack-
>Parameters.DeviceIoControl.OutputBufferLength;
  ULONG InputLength = pIoStack-
>Parameters.DeviceIoControl.InputBufferLength;

  PUCHAR buf = (PUCHAR)Irp->AssociatedIrp.SystemBuffer;

  OutputLength = 4;

  switch (ctlCode)
  {
   case IOCTL_READ:
    dx->L1 = dx->L1 * 3;
    RtlMoveMemory(buf,
                 (PUCHAR)&dx->L1,
                 OutputLength);
    Irp->IoStatus.Information = OutputLength;
    break;
  case IOCTL_WRITE:
```

```
    RtlMoveMemory((PUCHAR)&dx->L1,
                  buf,
                  InputLength);
    break;
  }

  Irp->IoStatus.Status = STATUS_SUCCESS;
  IoCompleteRequest(Irp, IO_NO_INCREMENT);
  return STATUS_SUCCESS;
}

NTSTATUS
DriverEntry(IN PDRIVER_OBJECT DriverObject,
            IN PUNICODE_STRING RegistryPath)
{
  PDEVICE_OBJECT DeviceObject;
  UNICODE_STRING NtDeviceName;
  UNICODE_STRING DosDeviceName;
  NTSTATUS status;

  RtlInitUnicodeString(&NtDeviceName, NT_DEVICE_NAME);
  status = IoCreateDevice(DriverObject,
                          sizeof(MY_DEVICE_EXTENSION),
                          &NtDeviceName,
                          FILE_DEVICE_UNKNOWN,
                          0,
                          FALSE,
                          &DeviceObject);
  if (!NT_SUCCESS(status))
   {
    IoDeleteDevice(DeviceObject);
    return status;
   }

  RtlInitUnicodeString(&DosDeviceName, DOS_DEVICE_NAME);
  status = IoCreateSymbolicLink(&DosDeviceName, &NtDeviceName);

  if (!NT_SUCCESS(status))
   {
    IoDeleteSymbolicLink(&DosDeviceName);
    IoDeleteDevice(DeviceObject);
    return status;
   }

  DeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;
  DeviceObject->Flags |= DO_BUFFERED_IO;

  DriverObject->MajorFunction[IRP_MJ_CREATE]  = Testw_Create;
  DriverObject->MajorFunction[IRP_MJ_CLOSE]   = Testw_Close;
  DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = Testw_IoctlRW;
```

```
 DriverObject->DriverUnload = Testw_Unload;

 return status;
}
```

В исходном тексте этого драйвера оп сравнению с предыдущим произошли некоторые изменения. Во-первых, добавлен код IOCTL-команды для записи (IOCTL_WRITE). Во-вторых, здесь изменился исходный текст функции-обработчика запроса IRP_MJ_DEVICE_CONTROL (эта функция называется Test_IoctlRW). Эта функция анализирует два кода – IOCTL_READ и IOCTL_WRITE, используя switch-оператор. Если приходит запрос IOCTL_WRITE, то 32-битовое значение записывается в переменную L1 из области расширения. При обнаружении кода IOCTL_READ значение переменной извлекается из переменной L1, умножается на 3 и возвращается в программу пользователя (оператор dx->L1 = dx->L1 * 3).

Вот каким будет окно работающей программы пользователя при работе с этим драйвером (рис. 7.12):

Рис. 7.12
Вид окна работающей программы

```
H:\WINDOWS\system32\cmd.exe
TESTW: output = 233
TESTW: input = output * 3 = 699
Для продолжения нажмите любую клавишу . . .
```

Таким образом, мы рассмотрели базовые концепции функционирования драйверов устройств операционных систем Windows. Перед тем как двигаться дальше, хочу обратить внимание читателей на еще один аспект программирования драйверов. Во многих случаях, особенно при разработке драйверов для электронных любительских устройств, управляемых от компьютера, отдельные фрагменты программного кода можно разрабатывать на встроенном ассемблере. Например, фрагмент программного кода, в котором выполняется умножение числа на 3 в драйвере testw.sys (оператор dx->L1 = dx->L1 * 3;) может быть заменен следующей последовательностью команд:

```
. . .
PUCHAR Lbuf = (PUCHAR)&dx->L1;

switch (ctlCode)
 {
  case IOCTL_READ:
    __asm {
```

```
          mov  ECX, DWORD PTR Lbuf
          mov  EAX, DWORD PTR [ECX]
          imul EAX, 3
          mov  DWORD PTR [ECX], EAX
         }
   . . .
```

На первый взгляд может показаться, что оператор

```
dx->L1 = dx->L1 * 3;
```

выполнится быстрее, но при дизассемблировании мы получим довольно большую группу команд ассемблера, представляющей код этого оператора, причем часть операций будут избыточными, либо не вполне оптимальными. Это не означает, что нужно использовать ассемблер везде и всюду, но при работе с портами ввода-вывода, особенно если важно быстродействие электроники устройства, ассемблер может очень даже быть полезным.

Перейдем к некоторым практически полезным применениям драйверов устройств при разработке устройств домашней электроники, управляемым от параллельного порта ПК.

7.5. Применение драйвера параллельного порта ПК

В практике любителей радио-, измерительной и робототехники наибольшей популярностью пользуется параллельный порт персонального компьютера. Многие устройства как созданные, так и разрабатываемые, управляются от параллельного порта. Изучив основы разработки драйверов, мы может без особого труда создать несложный драйвер для устройства, присоединенного к параллельному порту ПК. Сам по себе, без дополнительной электроники, параллельный порт может служить неплохим генератором звуковых колебаний, в качестве анализатора входных цифровых сигналов или использоваться для функционирования более сложных схем, например, аналого-цифровых преобразователей. Рассмотрим два аппаратно-программных проекта, работающих с параллельным портом принтера, для которых создадим простейшие драйверы.

Первое приложение представляет собой генератор прямоугольных импульсов частотой 500Гц. На основе этого примера читатели легко могут создать широкодиапазонный генератор от нескольких до 500 Гц. Прямоугольные импульсы вырабатываются на выходе регистра данных (адрес 0x378) параллельного порта, поэтому аппаратной части в смысле дополнительных микросхем здесь нет. Хотя, если вы планируете подключать выходы генератора к значительной нагрузке, то на выходах порта установите дополнительные буферные усилители-формирователи, способные обеспечить требуемый ток в нагрузке.

Для реализации программной части нам потребуется создать драйвер параллельного порта и приложение пользователя, работающее с этим драйвером. Начнем с драйвера. Самый простой метод генерации импульсов на каком-либо выводе порта – прочитать значение защелки порта, инвертировать его и переслать обратно в параллельный порт. Таким образом, можно получить один прямоугольный импульс. Для генерации непрерывной последовательности импульсов с определенной частотой необходимо периодически повторять операцию «считывание-инверсия-запись» в регистре 0x378. Период повторения импульсов можно задавать либо в самом драйвере устройства либо в приложении пользователя. Мы будем





использовать второй способ, задавая интервал времени в приложении пользователя – этот метод несравненно проще, чем тот, который можно было бы реализовать в самом драйвере.

Вначале разработаем драйвер (назовем его lptgen.sys), который будет работать с «виртуальным» устройством lptgen, в качестве которого фактически будет выступать порт вывода параллельного порта ПК. Наш драйвер будет обрабатывать IOCTL-команду IOCTL_WRITE в пакете запроса IRP_MJ_DEVICE_CONTROL. Вот исходный текст драйвера:

```
 #include <ntddk.h>

#define IOCTL_WRITE CTL_CODE(FILE_DEVICE_UNKNOWN,\
                             0x801,\
                            METHOD_BUFFERED,\
                            FILE_ANY_ACCESS)

#define NT_DEVICE_NAME L"\\Device\\lptgen"
#define DOS_DEVICE_NAME L"\\DosDevices\\lptgen"

#define DATA 0x378

VOID
lptgen_Unload (IN PDRIVER_OBJECT DriverObject)
{
  UNICODE_STRING DosDeviceName;

  RtlInitUnicodeString(&DosDeviceName, DOS_DEVICE_NAME);
  IoDeleteSymbolicLink(&DosDeviceName);
  IoDeleteDevice(DriverObject->DeviceObject);
}

NTSTATUS
lptgen_Create(IN PDEVICE_OBJECT DeviceObject,
             IN PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS;
  IoCompleteRequest(Irp, IO_NO_INCREMENT);
  return STATUS_SUCCESS;
}

NTSTATUS
lptgen_Close(IN PDEVICE_OBJECT DeviceObject,
             IN PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS;
  IoCompleteRequest(Irp, IO_NO_INCREMENT);
  return STATUS_SUCCESS;
}

NTSTATUS
lptgen_Ioctl(IN PDEVICE_OBJECT DeviceObject,
```




```
            IN PIRP Irp)
{
  PIO_STACK_LOCATION pIoStack = IoGetCurrentIrpStackLocation(Irp);
  ULONG ctlCode = pIoStack->Parameters.DeviceIoControl.IoControlCode;

  if (ctlCode == IOCTL_WRITE)
   {
   __asm {
         mov DX, DATA
          in  AL, DX
          not AL
          out DX, AL
        }
   }
  Irp->IoStatus.Status = STATUS_SUCCESS;
  IoCompleteRequest(Irp, IO_NO_INCREMENT);
  return STATUS_SUCCESS;
}

NTSTATUS
DriverEntry(IN PDRIVER_OBJECT DriverObject,
           IN PUNICODE_STRING RegistryPath)
{
 PDEVICE_OBJECT DeviceObject;
 UNICODE_STRING NtDeviceName;
 UNICODE_STRING DosDeviceName;
 NTSTATUS status;

 RtlInitUnicodeString(&NtDeviceName, NT_DEVICE_NAME);
 status = IoCreateDevice(DriverObject,
                        sizeof(MY_DEVICE_EXTENSION),
                        &NtDeviceName,
                        FILE_DEVICE_UNKNOWN,
                        0,
                        FALSE,
                        &DeviceObject);
 if (!NT_SUCCESS(status))
  {
   IoDeleteDevice(DeviceObject);
   return status;
  }

 RtlInitUnicodeString(&DosDeviceName, DOS_DEVICE_NAME);
 status = IoCreateSymbolicLink(&DosDeviceName, &NtDeviceName);

 if (!NT_SUCCESS(status))
  {
   IoDeleteSymbolicLink(&DosDeviceName);
   IoDeleteDevice(DeviceObject);
   return status;
```

```
  }

 DeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;
 DeviceObject->Flags |= DO_BUFFERED_IO;

 DriverObject->MajorFunction[IRP_MJ_CREATE]  = lptgen_Create;
 DriverObject->MajorFunction[IRP_MJ_CLOSE]   = lptgen_Close;
 DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = lptgen_Ioctl;
 DriverObject->DriverUnload = lptgen_Unload;

return status;
}
```

Во многом исходный текст драйвера нам знаком, поэтому я остановлюсь на реализации функции lptgen_Ioctl. Здесь мы используем блок ассемблерных команд. Вначале содержимое регистра 0x378 помещается в регистр AL командой in, где оно инвертируется, после чего выводится обратно в регистр по команде out. Здесь мы предполагаем, что в качестве базового регистра параллельного порта используется регистр с адресом 0x378. Если на вашем компьютере базовый адрес другой, например, 0x278, то необходимо переопределить константу DATA.

Таким образом, каждый раз при вызове функции DeviceIoControl() из приложения пользователя с кодом команды IOCTL_WRITE в драйвере будет выполняться инверсия выходов регистра 0x378.

Перейдем теперь к разработке приложения пользователя. Здесь, как и в предыдущих приложениях, мы будем использовать динамическую загрузку-выгрузку драйвера, а для операций с устройством использовать функцию WINAPI DeviceIoControl(), которой будем передавать код команды IOCTL_WRITE. Вот исходный текст пользовательской программы:

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>

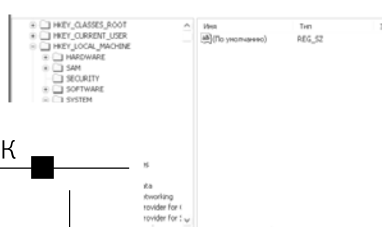
#define IOCTL_WRITE CTL_CODE(FILE_DEVICE_UNKNOWN,\
                              0x801,\
                            METHOD_BUFFERED,\
                            FILE_ANY_ACCESS)



SC_HANDLE scm, svc;
SERVICE_STATUS ServiceStatus;

void main(void)
{
  HANDLE fh;
  DWORD bytes;

  scm = OpenSCManager(NULL,
                     NULL,
                     SC_MANAGER_ALL_ACCESS);
  if (!scm)
```




```
 {
  printf("Cannot open SCM!\n");
  return;
 }

svc = CreateService(scm,
                 "lptgen",
                 "lptgen",
                 SERVICE_ALL_ACCESS,
                 SERVICE_KERNEL_DRIVER,
                 SERVICE_DEMAND_START,
                 SERVICE_ERROR_NORMAL,
                 "i:\\lptgen.sys",
                 NULL,
                 NULL,
                 NULL,
                 NULL,
                 NULL);
if (!svc)
 {
  printf("Cannot create service!\n");
  CloseHandle(scm);
  return;
 }

StartService(svc, 0, NULL);
CloseServiceHandle(svc);
CloseServiceHandle(scm);


fh = CreateFile("\\\\.\\lptgen",
               GENERIC_READ | GENERIC_WRITE,
               0,
               NULL,
               OPEN_EXISTING,
               0,
               NULL);
if (fh != INVALID_HANDLE_VALUE)
  {
  printf("Type any key to exit\n");
  while (! _kbhit())
     {
      DeviceIoControl(fh,
                   IOCTL_WRITE,
                   NULL,
                   0,
                   NULL,
                   0,
                   &bytes,
```

```
                     NULL);

       Sleep(1);
       };
     }

  }
  CloseHandle(fh);


  scm = OpenSCManager(NULL,
                  NULL,
                  SC_MANAGER_ALL_ACCESS);
  if (!scm)
   {
    printf("Cannot open SCM!\n");
    return;
   }
  svc = OpenService(scm, "lptgen", SERVICE_ALL_ACCESS);
  ControlService(svc, SERVICE_CONTROL_STOP, &ServiceStatus);
  DeleteService(svc);

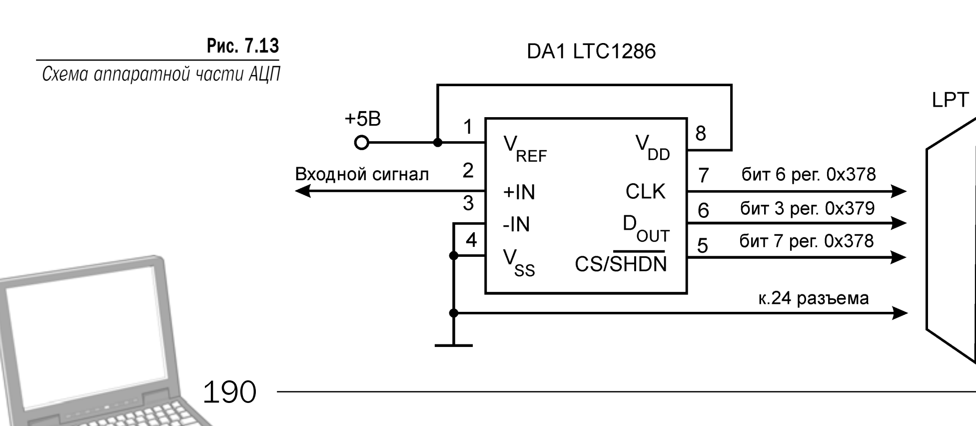
  CloseServiceHandle(svc);
  CloseServiceHandle(scm);

}
```

Здесь основная работа выполняется в цикле while (! _kbhit()). Этот цикл выполняется до тех пор (как и генерация сигнала на выходе параллельного порта), пока пользователь не нажмет любую клавишу. Функция Sleep(1) выполняет задержку на 1 миллисекунду, а поскольку для генерации полного периода колебаний требуется два цикла, то в результате получим период колебаний 2 мс, что соответствует частоте 500Гц. Для получения частоты, например, в 100Гц аргумент функции Sleep() должен иметь значение 5 и т. д.

Наш второй проект намного сложнее. Мы разработаем программную часть аналого-цифрового преобразователя на микросхеме LTC1286, который управляется с параллельного порта ПК. Этот проект, но с использованием драйвера PortTalk, мы анализировали в главе 3. Напомню, как выглядит аппаратная часть проекта (рис. 7.13):

Рис. 7.13
Схема аппаратной части АЦП





Сначала разработаем драйвер параллельного порта (назовем его adc1286.sys), который будет собирать данные с АЦП по запросу IRP_MJ_DEVICE_CONTROL и передавать их приложению пользователя. Исходный текст драйвера приведен далее:

```
#include <ntddk.h>

#define IOCTL_READ CTL_CODE(FILE_DEVICE_UNKNOWN,\
                           0x801,\
                         METHOD_BUFFERED,\
                         FILE_ANY_ACCESS)

#define NT_DEVICE_NAME L"\\Device\\adc1286"
#define DOS_DEVICE_NAME L"\\DosDevices\\adc1286"

typedef struct _MY_DEVICE_EXTENSION
{
      ULONG L1;
} MY_DEVICE_EXTENSION, *PMY_DEVICE_EXTENSION;

#define DATA 0x378
#define STATUS 0x379

VOID
adc_Unload (IN PDRIVER_OBJECT DriverObject)
{
  UNICODE_STRING DosDeviceName;

  RtlInitUnicodeString(&DosDeviceName, DOS_DEVICE_NAME);
  IoDeleteSymbolicLink(&DosDeviceName);
  IoDeleteDevice(DriverObject->DeviceObject);
}

NTSTATUS
adc_Create(IN PDEVICE_OBJECT DeviceObject,
          IN PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS;
  IoCompleteRequest(Irp, IO_NO_INCREMENT);
  return STATUS_SUCCESS;
}

NTSTATUS
adc_Close(IN PDEVICE_OBJECT DeviceObject,
         IN PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS;
  IoCompleteRequest(Irp, IO_NO_INCREMENT);
  return STATUS_SUCCESS;
```

```
}

NTSTATUS
adc_IoctlR(IN PDEVICE_OBJECT DeviceObject,
          IN PIRP Irp)
{
 PMY_DEVICE_EXTENSION dx = (PMY_DEVICE_EXTENSION)DeviceObject-
>DeviceExtension;
 PIO_STACK_LOCATION pIoStack = IoGetCurrentIrpStackLocation(Irp);

 ULONG ctlCode = pIoStack->Parameters.DeviceIoControl.IoControlCode;
ULONG OutputLength = pIoStack-
>Parameters.DeviceIoControl.OutputBufferLength;

 PUCHAR buf = (PUCHAR)Irp->AssociatedIrp.SystemBuffer;
 PULONG Lbuf = &dx->L1;
 dx->L1 = 0;

 OutputLength = 4;

if (ctlCode == IOCTL_READ)
 {
 __asm {

        push EBX
        mov  DX, DATA
        xor  AX, AX
        bts  AX, 7
        out  DX, AL
        btr  AX, 7
        out  DX, AL

        mov  BX, 15
     next:
        xor  AX, AX
        mov  DX, DATA
        btr  AX, 6
        out  DX, AL

        mov  DX, STATUS
        in   AL, DX
        bt   AX, 3
        rcl  CX, 1

        mov  DX, DATA
        bts  AX, 6
        out  DX, AL

        dec  BX
```




```
        jnz  next

        mov  DX, DATA
        bts  AX, 7
        out  DX, AL

        pop  EBX
        and  CX, 0x0FFF
        mov  EDX, DWORD PTR Lbuf
        mov  WORD PTR [EDX], CX
     }
  RtlMoveMemory(buf,
               (PUCHAR)&dx->L1,
              OutputLength);
 }
   Irp->IoStatus.Information = OutputLength;
   Irp->IoStatus.Status = STATUS_SUCCESS;
   IoCompleteRequest(Irp, IO_NO_INCREMENT);
   return STATUS_SUCCESS;
}

NTSTATUS
DriverEntry(IN PDRIVER_OBJECT DriverObject,
           IN PUNICODE_STRING RegistryPath)
{
  PDEVICE_OBJECT DeviceObject;
  UNICODE_STRING NtDeviceName;
  UNICODE_STRING DosDeviceName;
  NTSTATUS status;

  RtlInitUnicodeString(&NtDeviceName, NT_DEVICE_NAME);
  status = IoCreateDevice(DriverObject,
                         sizeof(MY_DEVICE_EXTENSION),
                         &NtDeviceName,
                         FILE_DEVICE_UNKNOWN,
                         0,
                         FALSE,
                         &DeviceObject);
  if (!NT_SUCCESS(status))
   {
      IoDeleteDevice(DeviceObject);
      return status;
   }

  RtlInitUnicodeString(&DosDeviceName, DOS_DEVICE_NAME);
  status = IoCreateSymbolicLink(&DosDeviceName, &NtDeviceName);

  if (!NT_SUCCESS(status))
   {
```

```
      IoDeleteSymbolicLink(&DosDeviceName);
      IoDeleteDevice(DeviceObject);
      return status;
   }

  DeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;
  DeviceObject->Flags |= DO_BUFFERED_IO;

  DriverObject->MajorFunction[IRP_MJ_CREATE]  = adc_Create;
  DriverObject->MajorFunction[IRP_MJ_CLOSE]   = adc_Close;
  DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = adc_IoctlR;
  DriverObject->DriverUnload = adc_Unload;

  return status;
}
```

В этом драйвере основная работа выполняется функцией adc_IoctlR, которая обрабатывает пакет запроса IRP_MJ_DEVICE_CONTROL с кодом IOCTL-команды IOCTL_READ. Поскольку драйвер должен возвращать данные приложению, то мы определили беззнаковую целочисленную 32-битовую переменную L1 в области расширения. Здесь нужно сделать одно очень важное замечание: драйвер ядра, как правило, обрабатывает целочисленные данные, хотя может работать и с функциями математического сопроцессора. В данном случае мы будем передавать в программу пользователя 32-битовый результат аналого-цифрового преобразования, а окончательную обработку результата будет выполнять приложение пользователя. В конечном итоге на экран дисплея будет выведено значение результата преобразования в формате вещественного числа с плавающей точкой.

Второй важный момент. При работе с АЦП LTC1286 мы используем ассемблерный код. Для такого класса приложений реального времени размер и эффективность программного кода имеют первостепенное значение, хотя в других случаях вполне возможно использование функций _inp() и _outp языка C++.

Обратите внимание, как из ассемблерного кода мы получаем доступ к переменной L1 – многие ошибки в работе драйвером при использовании ассемблерного кода связаны именно с неправильной адресацией внешних по отношению к ассемблерному блоку команд. Хоть это и необязательно, но желательно сохранять содержимое регистров в стеке командой push перед началом обработки данных и выталкивать их оттуда командой pop после окончания работы. Это особенно важно при разработке драйверов, в которых работает несколько программных потоков. В данном случае, мы сохраняем только содержимое регистра EBX.

Теперь перейдем к приложению пользователя, которое получает данные от нашего устройства adc1286. Исходный текст приложения показан далее:

```
#include <windows.h>
#include <stdio.h>

#define IOCTL_READ CTL_CODE(FILE_DEVICE_UNKNOWN,\
                            0x801,\
                          METHOD_BUFFERED,\
                          FILE_ANY_ACCESS)
```




```
SC_HANDLE scm, svc;
SERVICE_STATUS ServiceStatus;

void main(void)
{
  HANDLE fh;
  int binRes;
  float total;

  DWORD bytes;

  scm = OpenSCManager(NULL,
                    NULL,
                    SC_MANAGER_ALL_ACCESS);
  if (!scm)
   {
    printf("Cannot open SCM!\n");
    return;
   }

  svc = CreateService(scm,
                   "adc1286",
                   "adc1286",
                   SERVICE_ALL_ACCESS,
                   SERVICE_KERNEL_DRIVER,
                   SERVICE_DEMAND_START,
                   SERVICE_ERROR_NORMAL,
                   "i:\\adc1286.sys",
                   NULL,
                   NULL,
                   NULL,
                   NULL,
                   NULL);
  if (!svc)
   {
    printf("Cannot create service!\n");
    CloseHandle(scm);
    return;
   }

  StartService(svc, 0, NULL);
  CloseServiceHandle(svc);
  CloseServiceHandle(scm);


  fh = CreateFile("\\\\.\\adc1286",
                 GENERIC_READ | GENERIC_WRITE,
                 0,
                 NULL,
                 OPEN_EXISTING,
```

```
                0,
                NULL);
  if (fh != INVALID_HANDLE_VALUE)
    {

      DeviceIoControl(fh,
                      IOCTL_READ,
                      NULL,
                      0,
                      &binRes,
                      sizeof(binRes),
                      &bytes,
                      NULL);
      total = 5.0 / 4096.0 * binRes;
      printf("ADC LTC1286 result: %5.3f V\n", total);
  }
  CloseHandle(fh);


  scm = OpenSCManager(NULL,
                     NULL,
                     SC_MANAGER_ALL_ACCESS);
  if (!scm)
   {
    printf("Cannot open SCM!\n");
    return;
   }
  svc = OpenService(scm, "adc1286", SERVICE_ALL_ACCESS);
  ControlService(svc, SERVICE_CONTROL_STOP, &ServiceStatus);
  DeleteService(svc);

  CloseServiceHandle(svc);
  CloseServiceHandle(scm);
}
```

В этом приложении, как и в предыдущих, используется WINAPI функция DeviceIoControl(), которая посылает устройству команду IOCTL_READ. Драйвер устройства adc1286 возвращает 32-битовый двоичный результат, который нужно преобразовать в вещественное число с плавающей точкой. По этой причине в приложении объявлены две переменные:

```
int binRes;
float total;
```

Переменная binRes будет содержать двоичный результат преобразования, а переменная total – окончательное вещественное значение аналого-цифрового преобразования. Для 12-битового АЦП с напряжением смещения, равным 5В, значение результата в формате плавающей точки будет вычисляться следующим образом:

```
total = 5.0 / 4096.0 * binRes;
```

Наше приложение получает одно значение преобразования, но можно доработать его, организовав цикл с определенным интервалом сканирования по методике, схожей с той, что была использована в предыдущем проекте.

Для дальнейшего совершенствования ваших навыков по разработке драйверов Windows всем желающим могу посоветовать обратиться на сайт www.microsoft.com, на котором, помимо стандартной документации, имеются многочисленные статьи с описанием практических аспектов создания и отладки драйверов.

Заключение

В этой книге автор старался показать возможности персональных компьютеров при создании пользовательских электронных устройств, а также раскрыть некоторые программные технологии, помогающие разрабатывать такие устройства. В книгу, конечно же, не могли быть включены все аспекты проектирования таких систем, особенно с учетом бурного развития технологий, в частности беспроводных коммуникаций. Тем не менее, автор выражает уверенность, что анализ принципов создания и программирования таких аппаратно-программных систем принесет определенную пользу и при решении задач, которые не были рассмотрены на страницах данного издания. Автор надеется, что материал книги будет полезен заинтересованным читателям и станет незаменимым подспорьем для многих пользователей – как опытных, так и начинающих.

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС-КНИГА» наложенным платежом, выслав открытку или письмо по почтовому адресу: **123242, Москва, а/я 20** или по электронному адресу: **orders@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в Internet-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. **(495) 258-91-94, 258-91-95**; электронный адрес **books@alians-kniga.ru**.

Магда Юрий Степанович

Компьютер в домашней лаборатории

Главный редактор *Мовчан Д. А.*
dm@dmk-press.ru
Корректор Галушкина А. В.
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Подписано в печать 21.01.2008. Формат 70×100 $^{1}/_{16}$.
Гарнитура «Аббат». Печать офсетная.
Усл. печ. л. 18,75. Тираж 2000 экз.

Издательство ДМК Пресс
Web-сайт издательства: www.dmk-press.ru
Internet-магазин: www.abook.ru